

Using WinDbg and the debugger commands

Article • 11/14/2023

WinDbg is a debugger that can be used to analyze crash dumps, debug live user-mode and kernel-mode code, and examine CPU registers and memory.



This latest version features a more modern user experience with an updated interface, fully fledged scripting capabilities, an extensible debugging data model, built-in Time Travel Debugging (TTD) support, and many additional features.

For more information, see [WinDbg Overview](#).

To install the debugger, see [Install the Windows debugger](#).

To get started with WinDbg, see [Getting Started with Windows Debugging](#).

Debugger commands

To learn more about the debugger commands, see [Using Debugger Commands](#).

Feedback

Was this page helpful?

Yes

No

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

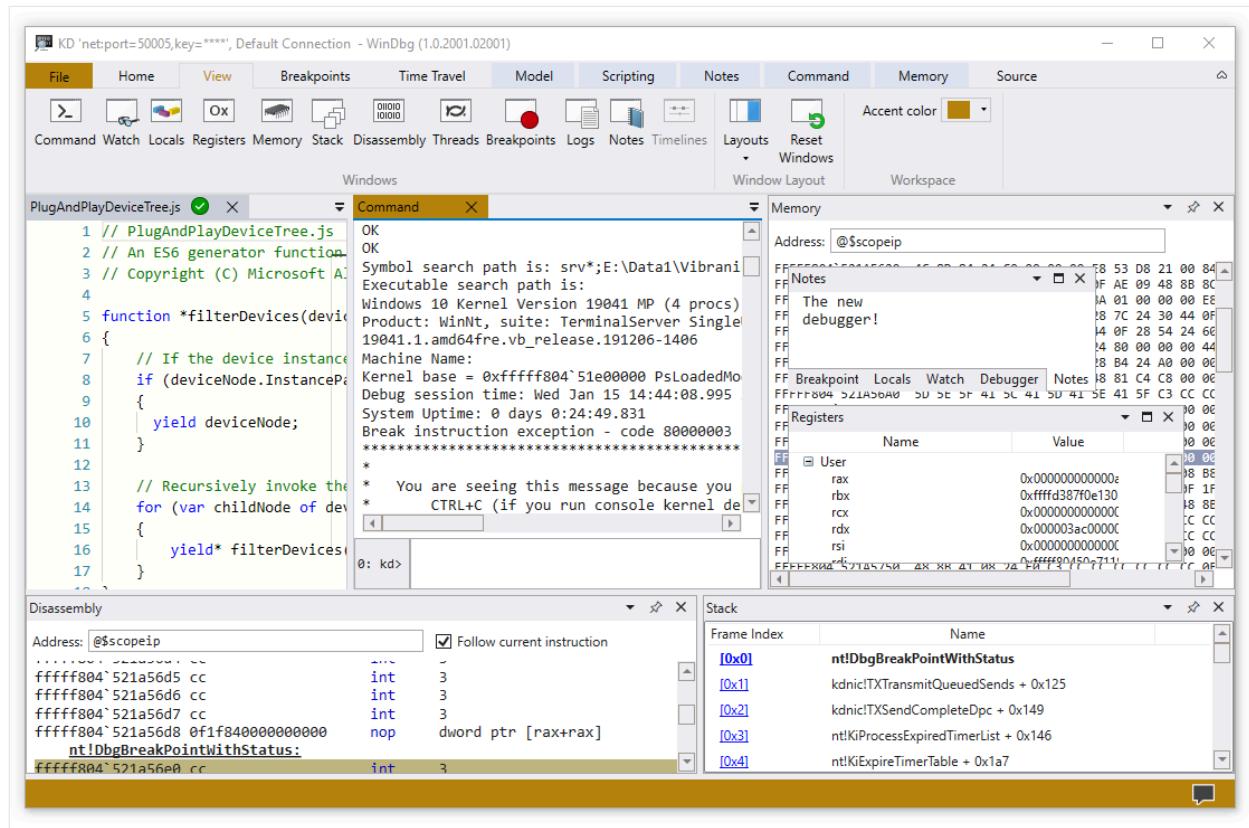
What is WinDbg?

Article • 08/26/2024

WinDbg is the latest version of WinDbg with more modern visuals, faster windows, a full-fledged scripting experience, built with the extensible debugger data model front and center.

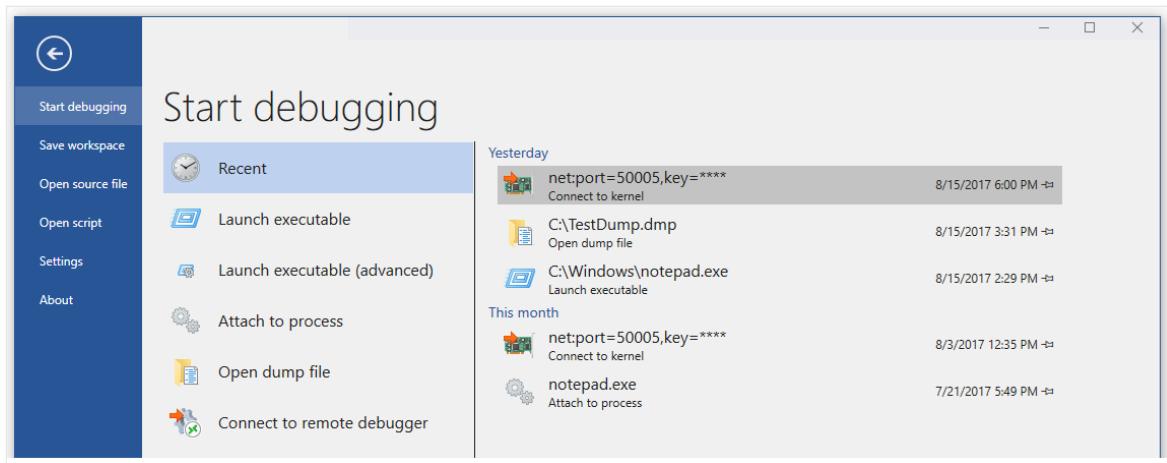
ⓘ Note

Formerly released as *WinDbg Preview* in the Microsoft Store, *WinDbg* leverages the same underlying engine as *WinDbg (Classic)* and supports all the same commands, extensions, and workflows.

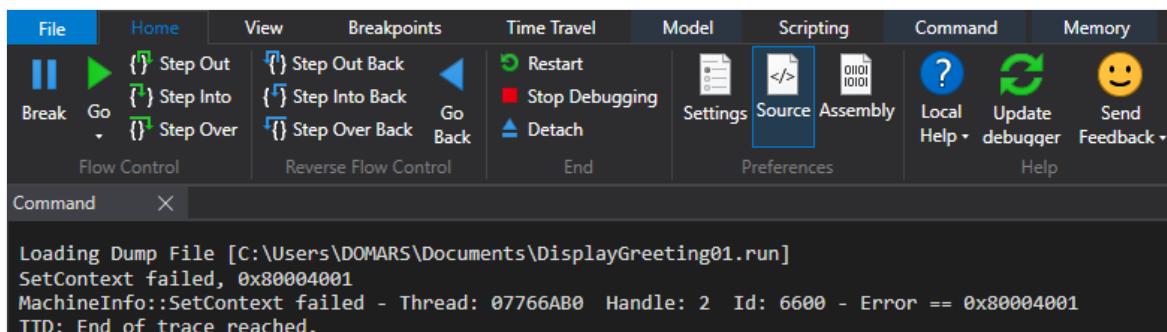


General features

- **Connection setup and recall** - Recent targets and session configurations are saved. They can be quickly restarted from the file menu.



- **Dark theme** - Go to File > Settings to enable the dark theme.



- **Keyboard navigation** - Use Ctrl+Tab to easily navigate between windows with just your keyboard.

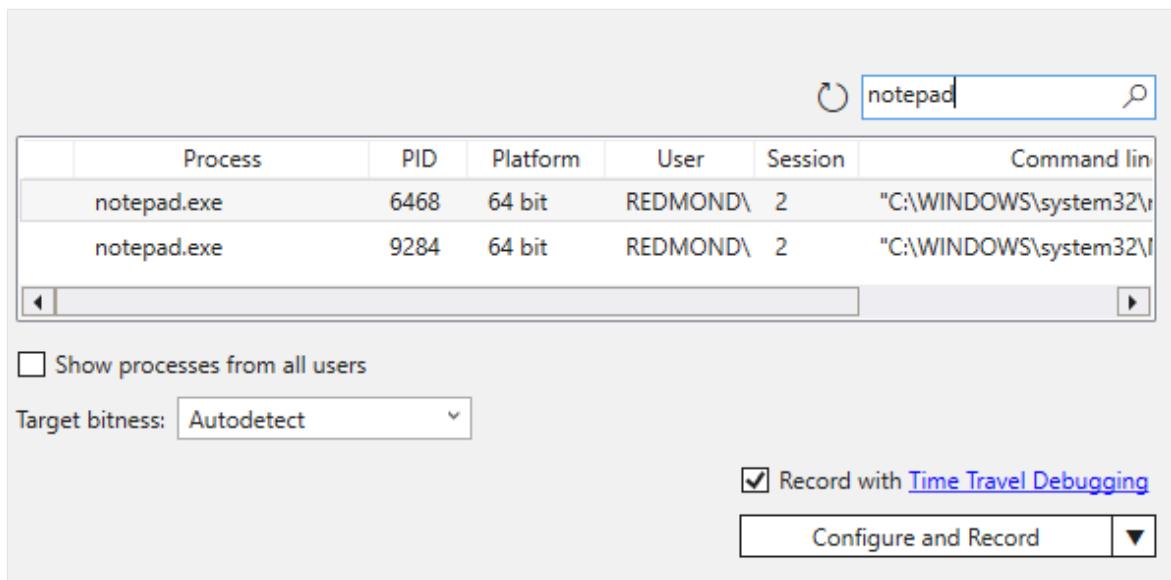


- **Dump file processor detection** - Autodetects processor architecture for easier managed debugging.
- **Performance improvements** - Tool windows load asynchronously and can be canceled. When you run a command, WinDbg can stop the loading of your locals, watch, or other windows.

Start debugging view

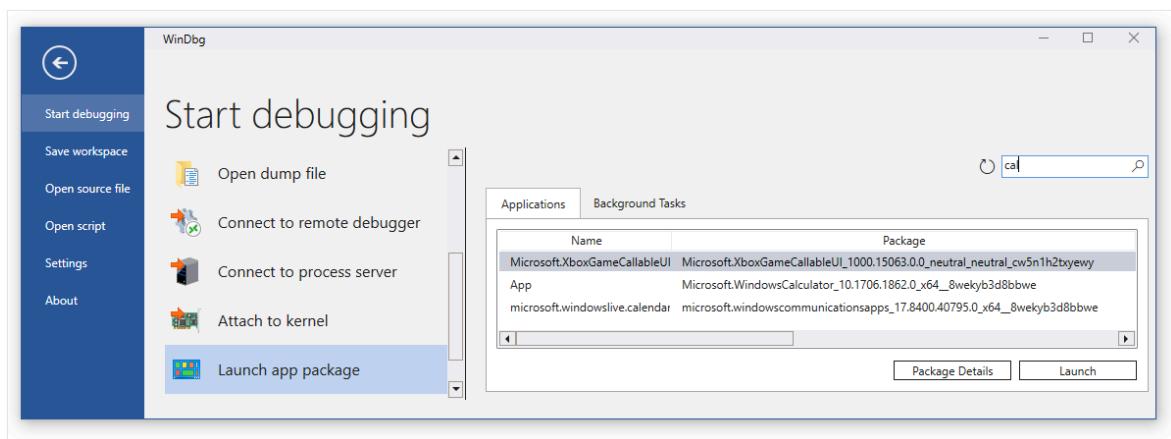
- **Integrated Time Travel Debugging (TTD)** - Use the "Record with Time Travel Debugging" checkbox when launching or attaching to a process. WinDbg will set up TTD, start recording, and open the trace afterwards.

For more information, see [Time Travel Debugging - Overview](#).



- **Launch App packages** - Debug your universal app or background task in a single click.

For more information, see [Launch App Package](#).



- **Attach to a process** - The new attach view provides a detailed view of running processes, easier configuration, and search support.

The screenshot shows the Windows Task Manager interface. At the top, there are tabs for 'File', 'Home', 'View', 'Breakpoints', 'Model', 'Scripting', 'Command' (which is selected), 'Memory', and 'Source'. Below the tabs, a list of processes is displayed:

Process	PID	Platform	User	Session	Command line
notepad.exe	27292	64 bit	REDMOND\domars	2	"C:\WINDOWS\system32\notepad.exe"
notepad.exe	26476	64 bit	REDMOND\domars	2	C:\Windows\System32\notepad.exe
ONENOTEM.EXE	14876	32 bit	REDMOND\domars	2	/tsr

Below the table, there are several configuration options:

- Show processes from all users
- Target bitness: Autodetect
- Record with Time Travel Debugging
- Attach dropdown menu

Improved tool windows

- Command** - The command window has improved DML support, text highlighting, search (including Regex).

The screenshot shows the Command window with the title 'Command'. The window displays a list of kernel modules, each with its address, name, and state (deferred). The list includes:

```

00007ffe`07400000 00007ffe`075c0000 MI_AIOINIT (deferred)
00007ffe`ba200000 00007ffe`ba284000 WINSPPOOL (deferred)
00007ffe`bb450000 00007ffe`bb604000 PROPSVVS (deferred)
00007ffe`beb20000 00007ffe`beb58000 IPHLAPI (deferred)
00007ffe`bef70000 00007ffe`bef7b000 CRYPTBASE (deferred)
00007ffe`bf000000 00007ffe`bf0a5000 bcrypt (deferred)
00007ffe`bf580000 00007ffe`bf5cc000 powerprof (deferred)
00007ffe`bf5f0000 00007ffe`bf601000 kernel_apcore (deferred)
00007ffe`bf610000 00007ffe`bf62f000 profapi (deferred)
00007ffe`bf630000 00007ffe`bf63a000 FILTLIB (deferred)
00007ffe`bf640000 00007ffe`bf64d000 windows_storage (deferred)
00007ffe`bf640000 00007ffe`c00d2000 gd132full (deferred)
00007ffe`c8190000 00007ffe`c8403000 KERNEL32 (deferred)
00007ffe`c8470000 00007ffe`c84b9000 cfgmgr32 (deferred)
00007ffe`c84c0000 00007ffe`c853a000 bcryptPrimitives (deferred)
00007ffe`c8540000 00007ffe`c863a000 wcrbase (deferred)
00007ffe`c8640000 00007ffe`c86df000 msvcp_win (deferred)
00007ffe`c86e0000 00007ffe`c8700000 win32u (deferred)
00007ffe`c8700000 00007ffe`c8b40000 SHELL32 (deferred)
00007ffe`c8b50000 00007ffe`c8be0000 msycrt (deferred)
00007ffe`c1bf0000 00007ffe`c1c99000 shcore (deferred)
00007ffe`c1e30000 00007ffe`c1ef2000 OLEAUT32 (deferred)
00007ffe`c1f60000 00007ffe`c1fb1000 SHLWAPI (deferred)
00007ffe`c1fc0000 00007ffe`c1fe8000 GDI32 (deferred)
00007ffe`c1ff0000 00007ffe`c20dd000 COMDLG32 (deferred)
00007ffe`c20e0000 00007ffe`c2403000 combase (deferred)
00007ffe`c2600000 00007ffe`c26b2000 KERNEL32 (deferred)
00007ffe`c26c0000 00007ffe`c27e4000 RPCRT4 (deferred)

```

- Source** - The source code window provides syntax highlighting and other general improvements similar to most modern text editors.

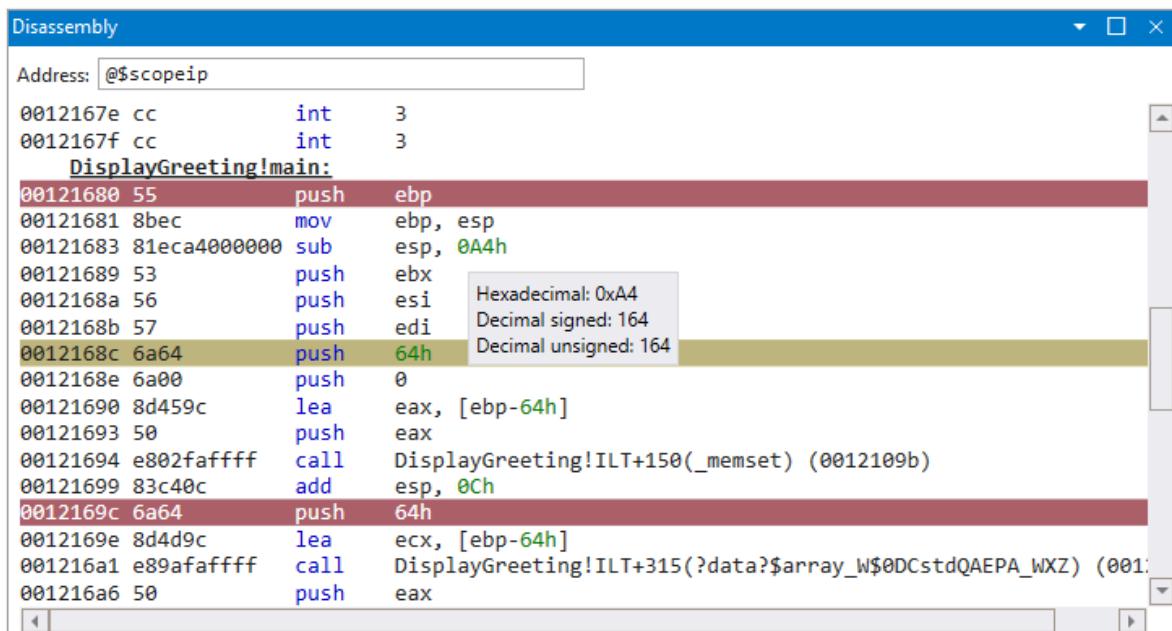


```

1 #include "stdafx.h"
2 #include <array>
3 #include <stdio.h>
4 #include <string.h>
5
6 void GetCppConGreeting(wchar_t* buffer, size_t size)
7 {
8     wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD
9
10    wcscpy_s(buffer, size, message);
11 }
12
13
14 int main()
15 {
16     std::array<wchar_t, 50> greeting{};
17
18     GetCppConGreeting(greeting.data(), sizeof(greeting));
19
20     wprintf(L"%ls\n", greeting.data());
21
22 }

```

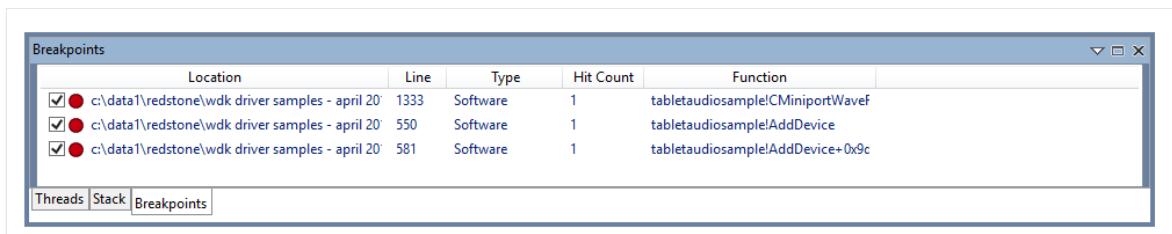
- **Disassembly** - The disassembly window is also improved, the highlight of the current instruction remains where it's when you scroll.



Address	OpCode	OpName	Reg	Value
0012167e	cc	int		3
0012167f	cc	int		3
00121680	55	push	ebp	
00121681	8bec	mov	ebp, esp	
00121683	81eca4000000	sub	esp, 0A4h	
00121689	53	push	ebx	
0012168a	56	push	esi	Hexadecimal: 0xA4 Decimal signed: 164
0012168b	57	push	edi	Decimal unsigned: 164
0012168c	6a64	push	64h	Decimal unsigned: 164
0012168e	6a00	push	0	
00121690	8d459c	lea	eax, [ebp-64h]	
00121693	50	push	eax	
00121694	e802faffff	call	DisplayGreeting!ILT+150(_memset)	(0012109b)
00121699	83c40c	add	esp, 0Ch	
0012169c	6a64	push	64h	
0012169e	8d4d9c	lea	ecx, [ebp-64h]	
001216a1	e89afaffff	call	DisplayGreeting!ILT+315(?data?\$array_W\$0DCstdQAEPA_WXZ)	(0012109f)
001216a6	50	push	eax	

- **Breakpoints** - The breakpoints window shows all your current breakpoints, a one-click toggle, and a hit count.

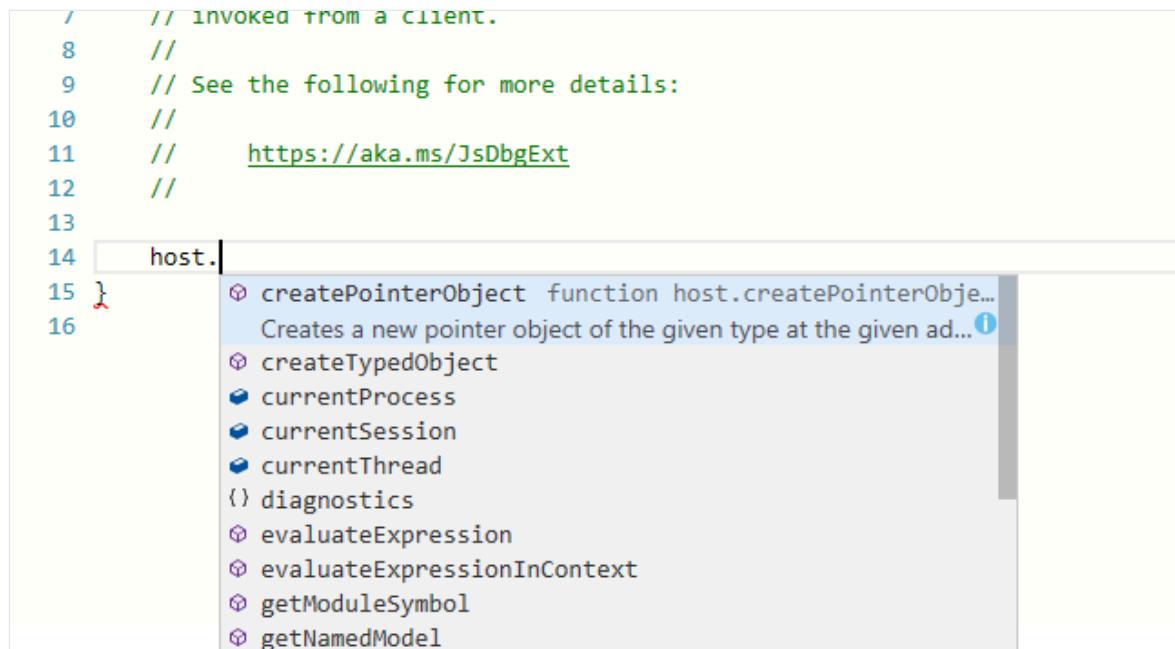
For more information, see [Breakpoints](#).



Location	Line	Type	Hit Count	Function
c:\data1\redstone\wdk driver samples - april 20\	1333	Software	1	tabletaudiosample!CMiniportWavef
c:\data1\redstone\wdk driver samples - april 20\	550	Software	1	tabletaudiosample!AddDevice
c:\data1\redstone\wdk driver samples - april 20\	581	Software	1	tabletaudiosample!AddDevice+0x9c

- **Scripting** - The new scripting window makes developing JavaScript and NatVis extensions easier, with error highlighting and IntelliSense.

For more information, see [WinDbg - Scripting](#).



```

7 // invoked from a client.
8 //
9 // See the following for more details:
10 //
11 //     https://aka.ms/JsDbgExt
12 //
13
14 host.
15 }
16

```

The screenshot shows a code editor with the following snippet:

```

7 // invoked from a client.
8 //
9 // See the following for more details:
10 //
11 //     https://aka.ms/JsDbgExt
12 //
13
14 host.
15 }
16

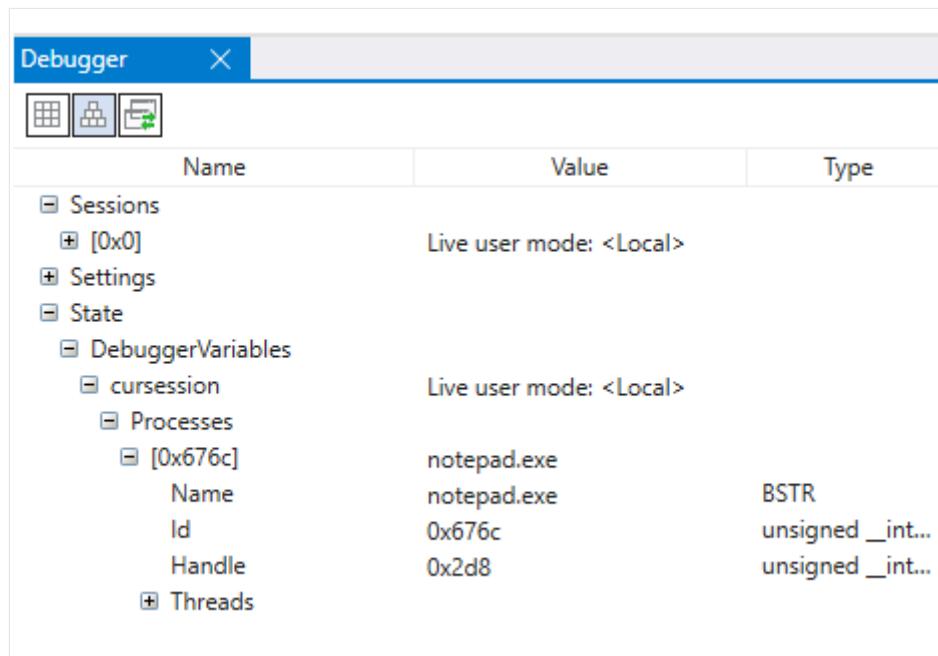
```

An IntelliSense tooltip is displayed over the word 'host.', listing various methods and properties:

- createPointerObject
- createTypedObject
- currentProcess
- currentSession
- currentThread
- diagnostics
- evaluateExpression
- evaluateExpressionInContext
- getModuleSymbol
- getNamedModel

- **Data model** - The model window provides an expandable and browsable version of `dx` and `dx -g`, letting you create powerful tables on-top of your NatVis, JavaScript, and LINQ queries.

For more information, see [WinDbg - Data model](#).



The screenshot shows the WinDbg Debugger window with the title bar "Debugger". Below the title bar is a toolbar with three icons: a grid, a list, and a search icon. The main area is a table with columns "Name", "Value", and "Type". The table displays the following data:

Name	Value	Type
Sessions		
[0x0]	Live user mode: <Local>	
Settings		
State		
DebuggerVariables		
cursession	Live user mode: <Local>	
Processes		
[0x676c]	notepad.exe	
Name	notepad.exe	BSTR
Id	0x676c	unsigned __int...
Handle	0x2d8	unsigned __int...
Threads		

- **Locals and watch** - The locals and watch windows are both based off the data model that is used by the `dx` command. This means they benefit from the same features as other data model windows.

- **Memory** - The memory window has highlighting and improved scrolling.
- **Logs** - This is an under the covers log of the WinDbg internals. It can be viewed for troubleshooting or to monitor long running commands.

Providing feedback

Your feedback helps our team guide WinDbg's development and prioritize features.

To report any bugs or suggest a new feature, you can follow the feedback button in the ribbon to go to the [GitHub page ↗](#) where you can file a new issue.

Other resources

- For information on what's new in the most recent release, see [Release notes](#).
- Review these topics to install and configure WinDbg:
 - [WinDbg – Command line startup options](#)
 - [WinDbg – Settings and workspaces](#)
 - [WinDbg – Keyboard shortcuts](#)
- These topics describe how to get connected to the environment that you want to debug:
 - [WinDbg – Start a user-mode session](#)
 - [WinDbg – Start a kernel mode session](#)
- Watch these episodes of the [Defrag Tools](#) show to see WinDbg in action:
 - [Defrag Tools #182](#) - Tim, Chad, and Andy go over the basics of WinDbg and some of the features.
 - [Defrag Tools #183](#) - Nick, Tim, and Chad use WinDbg and go over a quick demo.
 - [Defrag Tools #184](#) - Bill and Andrew walk-through the scripting features in WinDbg.
 - [Defrag Tools #185](#) - James and Ivette provide an introduction to Time Travel Debugging.
 - [Defrag Tools #186](#) - James and JCAB covers advanced Time Travel Debugging.
- Additional tips and tricks can be found in the [WinDbg blog archive](#).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Release notes

Article • 08/29/2024



This topic provides information on what's new in WinDbg. Earlier versions were released as *WinDbg Preview*.

1.2407.24003.0

New Features

New JavaScript Scripting Provider

We are now shipping a new backwards compatible JavaScript provider based on V8. As a result of this, arm64 hosts can now use JavaScript extensions. The arm64 UI also now defaults to the arm64 engine.

The new provider is enabled by default on all architectures. On x86 and x64, it may be turned on/off via the `.veighton` and `.veightoff` commands. At some point in the future, these commands and the legacy JavaScript provider will be removed.

UI Extensibility via The Extension Gallery and Data Model

Debugger extensions with extension gallery manifests can now add icons to the new Extensions tab of the WinDbg ribbon. Clicking these icons will trigger a visualization of a given data model query. This means of extensibility is accomplished by means of adding views to the extension gallery manifest. A view binds a coupling of a name and an icon to a data model query and view kind. Documentation can be found at [Extension Gallery Manifest Documentation](#)

Added Exception parameters to the Extension Gallery Exception load trigger

A gallery extension can now be conditionally loaded based on specific exception parameter values. See [Extension Gallery Manifest Documentation](#) for details.

Configuration option to never download specific symbols

You can now skip downloading symbols for specified images.

In the `Symbols` section of your `DbgX.xml` configuration file, you can add a list of symbols:

XML

```
<Namespace Name="DisabledSymbolsDownload">
  <Setting Name="MSEdge.dll" Type="VT_BSTR" Value=""></Setting>
</Namespace>
```

In the debugger, you can run `dx Debugger.Settings.Symbols.DisabledSymbolsDownload` to view the current list of symbols not being downloaded.

You can also change the list directly from the debugger like this:

```
dx @$curprocess.Modules[1].DisableSymbolDownload()
dx @$curprocess.Modules[1].EnableSymbolDownload()
```

New command to display SourceLink information

To aid debugging issues with downloading sources based on [embedded SourceLink](#), you can now use the command `!lmsourcesinfo module` to display the contents of the SourceLink that the debugger will use.

General UI Improvements

We've made some small improvements to windows like the modules window or the stack window:

- You can now sort grid views by clicking on the header column.
- There are more right-click actions available.
- The stack window shows frames as inlined when appropriate.

Bug fixes

- Fixed an issue where PDBs larger than 4GB would not be downloaded over HTTP.
 - Edge is an example of such a PDB.
- Fixed caching issue in the debugger credential provider (now it will cache credentials per site).

- Fixed bugs during evaluation of the Extension Gallery load triggers.
- Fixed a bug with `!address` that could cause a crash.
- `!heap -a -a` should work now.

TTD

See [TTD Release Notes](#) for version 1.11.410.

1.2402.24001.0

New Features

Live Linux debugging

You can now live debug a Linux process. For more information, see these topics:

[Linux live remote process debugging](#)

[Linux symbols and sources](#)

The WinDbg client for the foreseeable future will still require Windows, as WinDbg does not run on Linux.

Record a subset of execution with Time Travel Debugging

You can now specify a list of modules to record to focus TTD on the modules you care about. This can substantially reduce recording overhead. To use this simply check "Record subset of execution" in the "Configure and Record" dialog box, then specify the module(s) you want recorded. (see [known issues](#) below).

For finer control, you can precisely record your program using a new in-process recording API. See how with our [sample code](#) and [documentation](#).

Compact register window

The new register window looks just like the output from the `r` command. Note that editing from the new register window is not yet supported.

New parameters for `e` (Enter Values) into memory commands

The `e` (Enter Values) into memory commands now support physical addresses just like the `d` (Display Memory) commands. Note these options are only supported in kernel mode:

[+] Expand table

Option	Description
<code>/p</code>	Uses physical memory addresses for the display. The range specified by Range will be taken from physical memory rather than virtual memory.
<code>/p[c]</code>	Same as <code>/p</code> , except that cached memory will be read. The brackets around c must be included.
<code>/p[uc]</code>	Same as <code>/p</code> , except that uncached memory will be read. The brackets around uc must be included.
<code>/p[wc]</code>	Same as <code>/p</code> , except that write-combined memory will be read. The brackets around wc must be included.

For more information, see [e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza \(Enter Values\)](#).

Bug Fixes

- Support for Rust name demangling in Linux debugging
- Improvements to CLR debugging
 - Add the ability to force mixed mode stack walking CLR frames
 - Added `ForceMixedModeStackWalker` setting added to `config.xml` or `DbgX.xml`.
 - Added `!forceclrmixedmodeunwind` command for one-off investigations or if you can't change the config setting.
- General improvements of CLR debugging on Linux
- Fix a number of issues with respect to `LinuxKernel.js` and per-cpu variables
 - Added `!runq` command to `LinuxKernel.js` to be able to dump the per-cpu kernel scheduler run queues (similar in form to the Linux crash utility's `runq` command)
 - Fixed various commands in `LinuxKernel.js` to correspond to kernel changes: in more recent kernels, timer tree uses cached RB nodes instead of raw RB nodes; in more recent kernels, the `all_bdevs` list is gone, so block devices are now walked from the superblock instead (`blockdev_superblock`)
- Various natvis fixes to work more like recent versions of Visual Studio
- Prefer sourcelink vs legacy sourceinfo
 - Added `!lmsourcesinfo <module>` command to display sourcelink streams from pdb (to aid debugging issues with sourcelink).

Known issues

When specifying a list of modules to record with TTD, specifying more than one module doesn't work in this release.

1.2308.2002.0

Many bug fixes and small improvements.

1.2306.12001.0

Many bug fixes and small improvements.

1.2303.30001.0

Note

The debugger has been renamed from *WinDbg Preview* to *WinDbg*.

The legacy debugger released with [Debugging Tools for Windows](#) will be referenced as *WinDbg (classic)*.

Time Travel Debugging on ARM64

Not only does WinDbg run natively on ARM64, we also enabled Time Travel Debugging (TTD) of native ARM64 processes as well. You can now record native ARM64 processes (and x86 and ARM32 processes) and debug them easily, even on your x64 machine. Support for x64 or x64-enabled (ARM64EC) processes on ARM64 devices isn't yet available.

Overall TTD performance has also been improved, with recording overhead substantially reduced for processes that use common SIMD instructions. In total 65 new instructions were optimized, ranging from MMX through SSE 4.2, AVX and AVX2.

Ambiguous Breakpoint Resolution

Ambiguous breakpoints allow for the debugger to set breakpoints in certain scenarios where a breakpoint expression resolves to multiple locations. For more information, see [Ambiguous breakpoint resolution](#).

1.2107.13001.0

Regex search

WinDbg Preview search now includes the option to search using regular expressions - Regex. Enable Regex in the window you'd like to search by doing Ctrl+F, then toggle the button labeled `.*` next to the search box.

Restricted Mode

Restricted Mode is an optional setting that can limit the type of debugging sessions WinDbg Preview can start to remote debugging sessions and dump files only. For more information, see [WinDbg - Restricted Mode](#).

Shadowed variable support

In past versions, WinDbg throws "Ambiguous Symbol" errors when trying to evaluate `(??)` or display `(dx)` a variable that shares its name with another variable in scope.

Windbg will now disambiguate variables that share the same name by appending `@n` to the variable name. For example: `foo @0`, `foo @1`

This change will also be reflected in the Locals window. Previously, only the nearest variable in scope would be displayed.

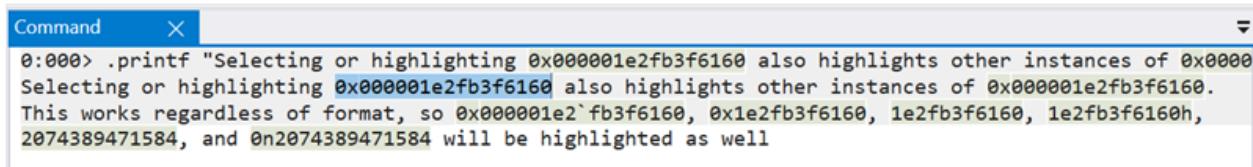
1.2104.13002.0

Smart number selection and search

A convenient productivity feature of WinDbg Preview is the ability to detect patterns in the command window. To do this, select any text, and all other instances of that text highlighted. Because this highlighting is useful when looking at memory patterns, it now will also highlight equivalent numbers in other radices, no matter how the number is formatted in hex, decimal, or scientific notation. For more information about numbering schemes, see [n \(Set Number Base\)](#).

Example:

When selecting `0x000001e2fb3f6160`, all other instances are highlighted no matter the format.



```
Command X
0:000> .printf "Selecting or highlighting 0x000001e2fb3f6160 also highlights other instances of 0x0000
Selecting or highlighting 0x000001e2fb3f6160 also highlights other instances of 0x000001e2fb3f6160.
This works regardless of format, so 0x000001e2`fb3f6160, 0x1e2fb3f6160, 1e2fb3f6160, 1e2fb3f6160h,
2074389471584, and 0n2074389471584 will be highlighted as well
```

This feature works with semi-temporary highlights as well. Ctrl + Double Click on a number to highlight all its instances. You can keep track of multiple important numbers throughout your command history this way (to clear the highlight, Ctrl + Double Click on the number again). Finally, this feature also works when searching numbers with Ctrl + F.

Source Code Extended Access

The source path command [.srcpath](#), [.lsrcpath \(Set Source Path\)](#) has been updated to include a new tag – *DebugInfoD*. For more information, see [Source Code Extended Access](#).

Host and guest states of WOW processes in the data model

When debugging a 32-bit WOW process from a 64-bit context, users can now access both the host and guest states within the data model.

32 bit guest state examples:

```
dx @$curprocess.Environment
```

```
dx @$curthread.Environment
```

```
dx @$curthread.Stack
```

64 bit host state examples:

```
dx @$curprocess.NativeEnvironment
```

```
dx @$curthread.NativeEnvironment
```

```
dx @$curthread.NativeStack
```

Javascript Debugging Improvements

Javascript loaded in the UI can now be directly debugged within the console using the [.scriptdebug](#) command. For more information, see [JavaScript Debugger Scripting -](#)

Accessibility improvements

With WinDbg Preview we are committed to building a debugger that is inclusive to engineers with disabilities, we are continuously improving accessibility. The following improvements have been made.

- Command window links can now be clicked via the keyboard (Shift+Enter)
- Improved keyboard navigation of main menu
- Improved keyboard navigation of ribbon
- Increased contrast on UI elements

New “Overwrite” data access type for Time Travel Debugger

Time Travel Debugger (TTD) now provides an “Overwrite” data access type. Memory queries such as `dx @$cursession.TTD.Memory()` now have an additional column showing the old values of writes.

Other fixes, improvements, and updates

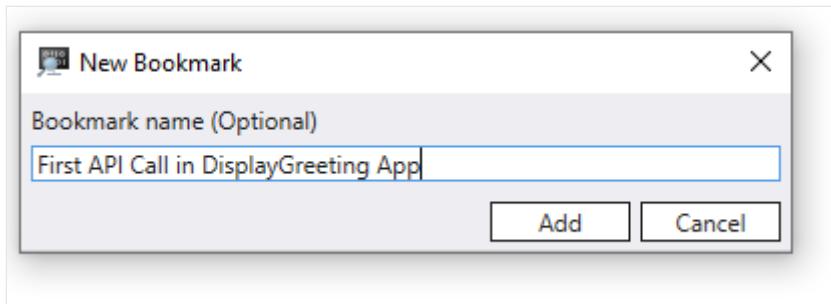
- Added feature to automatically detect and apply workaround for USB 3.1 hardware issue when both kernel debugging host and target are USB 3.1 controllers.
- Added a new UI shortcut: Ctrl + Shift + Click over a DML link will copy it to the clipboard

1.0.2007.01003

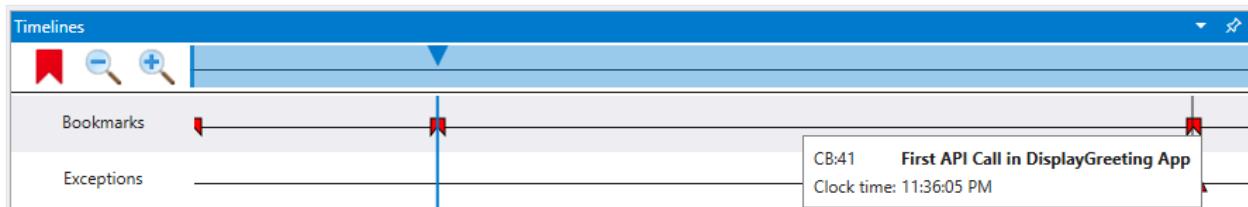
Timeline Bookmarks

Bookmark important Time Travel positions in WinDbg instead of manually copy pasting the position to notepad. Bookmarks make it easier to view at a glance different positions in the trace relative to other events, and to annotate them.

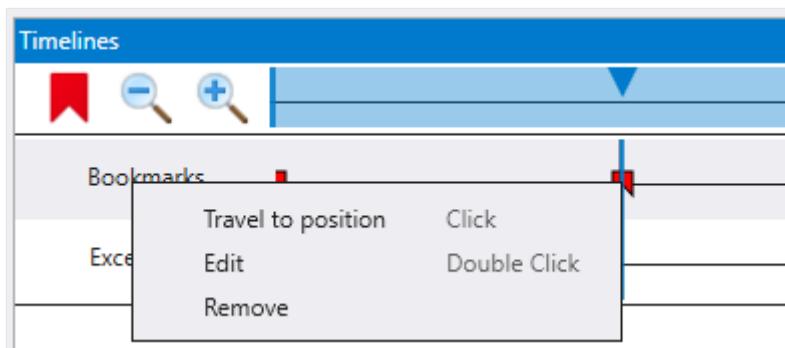
You can provide a descriptive name for bookmarks.



Access Bookmarks via the Timeline window available in *View > Timeline*. When you hover over a bookmark, it will display the bookmark name.



You can select and hold (or right-click) the bookmark to travel to that position, rename or delete the bookmark.



Modules Window

A new windows shows modules and their related information, it is available via the View ribbon. It displays:

- The name of the module including the path location
- The size in bytes of the loaded module
- The base address that the module is loaded at
- The file version

Name	Size	Base Address	File Version
E:\Data1\Vibraniun\Debugger\Timeline\Disp...	0x1e000	0x50000	
C:\WINDOWS\SYSTEM32\apphelp.dll	0x9f000	0x6b4a0000	10.0.18362.1
C:\WINDOWS\System32\KERNEL32.DLL	0xe0000	0x74af0000	10.0.18362.10022
C:\WINDOWS\System32\KERNELBASE.dll	0x1fc000	0x756e0000	10.0.18362.10024
C:\WINDOWS\SYSTEM32\ntdll.dll	0x19a000	0x76fd0000	10.0.18362.10024

Thread names/descriptions available in live debugging

Thread names that are set from SetThreadDescription are now available when doing live user-mode debugging. Thread names are available using the “~” command or the debugger data model.

```
dbgconsole

0:000> ~
 0  Id: 53a0.5ffc Suspend: 1 Teb: 000000b1`db1ed000 Unfrozen "Hello
world!"
 7  Id: 53a0.9114 Suspend: 1 Teb: 000000b1`db1ef000 Unfrozen
 8  Id: 53a0.2cc4 Suspend: 1 Teb: 000000b1`db1f1000 Unfrozen
 9  Id: 53a0.5c40 Suspend: 1 Teb: 000000b1`db1f3000 Unfrozen

0:000> dx @$curthread
@$curthread : ConsoleTestApp!ILT+25(mainCRTStartup)
(00007ff7`fac7101e)  [Switch To]
  Id          : 0x5ffc
  Name        : Hello world!
  Stack
  Registers
  Environment
```

Portable PDB support

Portable PDB support has been added. The Portable PDB (Program Database) format describes an encoding of debugging information produced by compilers of Common Language Infrastructure (CLI) languages and consumed by debuggers and other tools. For more information, see [Portable PDB Symbols](#).

Other changes and bug fixes

- WinDbg now supports AMD64 and Linux kernel dump debugging.
- Time travel recording enhancements and other fixes.

1.0.1912.11001

TTD Timelines - We've added a new window that displays a visual representation of important events in your trace: exceptions, breakpoints, function calls, and memory accesses. Timelines will automatically open and display exceptions (if present) and breakpoints. For more information, see [WinDbg Preview - Timeline](#).

Switched to default window chrome - The custom window chrome we were using, while prettier, was causing some scaling and resizing issues for a notable number of people, so we've opted to remove it for the time being.

File menu improved keyboard navigation - The file menu is now much easier to navigate with just a keyboard.

Other changes and bug fixes

- The stack and locals window will now be disabled when your target is running and won't show "Unspecified error" when there is no target.
- Added a "Services" column to the attach dialog to easily find which services are running.
- Fixed a bug that caused architecture detection to not work when launching applications with arguments.
- The disassembly window has improved disassembly when private symbols are loaded.
- jsprovider.dll is now loaded automatically, so we removed the "Load JSProvider" button from the scripting ribbon.

1.0.1908.30002

Improvements to TTD Calls objects - [Calls queries](#) now include parameter names, types, and values. When querying across traces for function calls you can get fully typed parameters and their values making it easy to filter down results by parameters.

Support for Open Enclave - WinDbg Preview can now debug Open Enclave (OE) applications for more information, see [Open Enclave debugging](#).

ELF Core Dumps - As part of supporting Open Enclave, WinDbg can open ELF core dumps and binaries as well as DWARF symbols (DWARF 5 is not currently supported) from both Enclaves and Linux applications. When opening a core dump from a non-Windows application, basic windows and commands should all work properly, but most extensions and Windows-specific commands will not work. ELF and DWARF files will be downloaded from symbol servers following the [key conventions defined here ↗](#). Enclaves are the only supported scenario, but we're open to feedback on opening other Linux core dumps.

TTD File format change - We've made a major update to the file format for TTD traces that breaks forward compatibility. Previous versions of WinDbg Preview will not be able to open traces recorded with this (and future) versions of WinDbg Preview, but this (and future) versions will be able to open both new and old traces.

Other changes

- TTD will now use the 64-bit engine for indexing and the appropriate debugger engine bitness for replays to minimize potential memory issues when indexing and

SOS issues when replaying.

- Running 'dx' without any parameters will now show the root namespace for easier browsability.
- You can now modify the default symbol and source cache location via the settings menu.
- Improved support for recording AVX-512 (recording of AVX-512 will cause a larger than normal slow-down).
- We've enabled [offline licensing](#).

1.0.1905.12001

Improvements to SymSetDiaSession error mitigation - Our fix last month to mitigate the error caused by applications injecting DbgHelp into our process was still not working in some scenarios. We've made improvements to it and will continue to monitor feedback on this error.

Accent color customization - A lot of scenarios need several instances of WinDbg open, and moving back and forth between them can be confusing and take some time to figure out which one is the "right" one. We've added the ability to change the blue accent color to help visually distinguish sessions and make swapping between them easier.

Just select the **View** ribbon and select an option for **Accent color** in the last section. When future sessions are launched from recent targets, the accent color will be persisted as part of the target's workspace.

Source tokenization improvements - The source window now has basic support for tokenizing Rust source files and C++ SEH __try/__except/__finally/__leave.

Croutine improvements - Improved support for coroutine local variables and certain optimized variables.

Default symbol and source cache setting - Added an option to the settings menu under **Debugging settings** to change the cache location for symbols. **Note** - There's a known issue that making this blank will cause source loading to fail. We'll be adding validation to prevent this from happening in a future release.

-pv fixes - Fixed a bug that may have prevented -pv (non-invasive attach) from working in some circumstances.

1.0.1904.18001

Fix for SymSetDiaSession error - We've had reports for a while of an error that prevents WinDbg Preview from being launched in some situations. There are a few external applications that attempt to inject a version of DbgHelp into our process before we load it. Some of them are using a version of DbgHelp with missing functionality, which causes this error when we attempt to use those features. We've added a fix for this and will be tracking if there are still scenarios in which it occurs.

Font control - We've added settings for controlling font and font size. There are two different settings, one for text windows (mono-spaced windows like disassembly, source, command, etc.) and one for tool windows (locals, stack, etc.). There's still a few areas that aren't affected by these options that we'll be updating in the future.

Highlighting improvements - Persistent highlighting of text in the command window will now also highlight text in the source and notes windows.

Source loading improvements - We've changed how loading source files works. Previously when opening a source file, engine operations like running additional commands weren't possible or were unpredictable. We've changed where the loading occurs to enable better parallelism and more reliable cancellation of source opening operations.

Other changes and bug fixes:

- Added "Go to disassembly" to the context menu of the source window.
- Added a checkbox to "Follow current instruction" in disassembly window.
- Fixed a bug that caused the command window to perform slowly when outputting lots of text.
- Changed page up and page down keys to perform similar to Visual Studio.
- When an ASM file is opened in the source window it will now have basic comment, string, and directive highlighting

1.0.1812.12001

This version includes these updates.

Debugger data model C++ header - There is a new C++ header, DbgModel.h, included as part of the Windows SDK for extending the debugger data model via C++. You can find more information in [Debugger Data Model C++ Overview](#). This release includes a new extension that adds some more "API style" features to the debugger data model that can be accessed through the 'dx' command, JavaScript, and the new DbgModel.h header. This extension extends the data model to include knowledge about assembly

and code execution through the [Debugger.Utility.Code](#) namespace, and the local file system through the [Debugger.Utility.FileSystem](#) namespace.

Synthetic types extension With this new API extension, we have a new sample up on our GitHub repo here - <https://github.com/Microsoft/WinDbg-Samples/tree/master/SyntheticTypes>. This JavaScript extension reads basic C header files and defines synthetic type information for the structures and unions defined in the header. Through the dx command, memory can then be viewed structured as if you had a PDB with type information for those types.

Other changes and bug fixes:

- WinDbg Preview will now more intelligently handle bringing source windows or the disassembly window to the foreground when stepping.
- Re-arranged WinDbg Preview's window title to have more important information at the start when kernel debugging.
- The alternating background contrast in the command window should be slightly more noticeable.

1.0.1810.2001

This version includes these updates.

- New Settings dialog that is accessed from the File menu or the Home ribbon.
- Events and exceptions settings dialog. This menu changes how the debugger handles events and exceptions, the equivalent of the 'sx' commands or WinDbg's event filters dialog. Select **Settings** on the home ribbon, then hit "Events and Exceptions" on the left to manage those.
- Improved TTD indexer with better performance. This increases the performance of indexing TTD trace files, making the indexing process much faster (between 2x-10x) while making index files much smaller (~50% smaller). The perf improvements are most noticeable for traces over 4GB in size, or when using a machine with many CPU cores (8+). The new indexer makes it more feasible to debug very large traces (50GB+).
- New *debugArch* launch flag for specifying architecture. WinDbg Preview attempts to launch the debugger engine with the correct bitness to the target to better support debugging managed code. There are circumstances where it can't determine the right bitness or you may want to override what it decides. Use --debugArch x86|amd64 to control the architecture of debugger engine.

Other changes and bug fixes:

- Fixed a bug that would cause black bars to appear on a full screen debugger with a floating window open.
- Fixed a bug that would cause symbol options to be cleared unintentionally.
- Command history is now preserved when launching from recent targets.
- In the data model window, you can now edit values.
- Un-indexed TTD traces will now be more clear that they're un-indexed.
- Improved performance of the locals window
- Added a ribbon button to save the command window logs to a file.
- Added .SelectMany(<projection>) to the default set of LINQ methods.

1.0.1807.11002

This version includes these updates.

Automatic saving and loading of breakpoints. This is a first step to replace workspaces. We're starting down that route by enabling the saving and loading of breakpoints. Launching something you've debugged previously from the "Recents" tab in the file menu will now load the breakpoints from that session. The plan is to expand this functionality to preserve more information across sessions. Hardware breakpoints (ba) and other various properties on breakpoints like thread and process specific contexts as well as conditions are not currently being saved.

Minor changes and bug fixes:

- Added command-line options -x, -xe, -xd, -xn, and -xi for controlling the handling of exceptions and events. These command-line options behave just like their command counter-parts.
- The notes window now supports bold, underline, and italics formatting.
- Fixed some zoom and scrolling issues.
- Selecting text in the command, memory, sources, or disassembly windows will now show a light highlight over other instances of the selected text.
- Fixed a bug where interrupting symbol loading would cause symbol loading to fail for the rest of the session.
- NatVis will now reload properly on restarting a session.

1.0.1805.17002

This version includes these updates.

New disassembly window - The disassembly window now includes:

- Scrolling up or down will continuously load more disassembly whenever possible.

- Syntax highlighting for numbers, code addresses, and opcodes.
- Selecting a code symbol will jump the disassembly window to that location.
- Hovering over numbers will show a tooltip that converts that number to other bases.
- Headers signifying the start of a function.

Faster source window - The source window has been updated to be faster and more resource efficient.

Minor changes and bug fixes:

- Fixed issues around symbol caching
- Fixed some cases where toggle initial break wasn't usable when the target isn't broken in
- If you hit tab in the command window with nothing available, the cursor will now stay in the input field
- WinDbg Preview will now auto-detect bitness when opening CAB files

1.0.1804.18003

This version includes these updates.

Symbol status and cancellation improvements - There are times where the debugger displays *BUSY* loading symbols and it's difficult to determine what it's doing and why without !sym noisy enabled. We've updated WinDbg Preview to have some better communication around what it's doing when loading symbols to help troubleshoot any issues. In addition to easily seeing exactly what's happening, we've made some changes that should make cancelling symbols more reliable and the Logs window will contain some of the details that's normally output when !sym noisy is enabled. If you hit View -> Logs you'll get the full noisy symbol loading output without having to turn it on and attempt to reload the symbols.

Experimental notes window - WinDbg Preview now has a window for taking notes. Just hit View -> "Notes" to open it. If you copy/paste into it, DML links will be preserved and still work as if it was the command window. You can also save and load notes files from the "Notes" ribbon when the window is open.

Experimental faster source window - To help improve the performance of WinDbg Preview there is an experimental new source window that is quite a bit more efficient. There's still a few gaps around context menus and syntax highlighting, but we want to give everyone the option of trying it out before it's finished to give us early feedback. Run \$UseFastSourceWindow to use it. If you want to go back to the old one, run

`$UseMonacoSourceWindow`. The setting will preserve across sessions, you will need to close and re-open source windows to get the new version.

JSProvider API version 1.2 - For JavaScript extensions that declare support for API version 1.2:

- Any object with a `.compareTo` method which exists the script will have a custom comparator on it (comparison operators will work in the DX evaluator and elsewhere: e.g.: `IModelObject::Compare`)
- Any object with a `.equals` method which exists the script will have a custom equality operator on it (`==` and `!=` will work in the DX evaluator and elsewhere: e.g.: `IModelObject::IsEqualTo`)
- Native or data model objects which enter the script will have `.compareTo` and `.equals` on them which allow access to any custom comparator or custom equality implementations.

Minor changes and bug fixes:

- `.server` will now list fully qualified domain name for easier use when there's domain issues around short names.
- Ctrl+G now works in the source window.
- Added address bar to the disassembly window.
- WinDbg Preview will now handle `_NT_SYMBOL_PATH` in a more expected way.
- Added `-server` command-line option.
- TTD data model queries can now be displayed progressively, so if you interrupt it you'll still see some results. This feature is still experimental and optional. Run `dx @$cursession.TTD.AsyncQueryEnabled = 1` to enable it.
- The 'dps' command now has links to the source files it refers to.

1.1801.19001.0

This version includes these updates.

Text Highlighting - You can now highlight all instances of selected text directly in the debugger. To use this feature, just select some text in the command window and then select "Highlight" in the command ribbon or hit CTRL+ALT+H. Using one of those on already highlighted text will remove the highlighting.

If you prefer using commands, you can use the "`$hl`" command:

`$hl ["someValueHere"]` - Highlight give text (or un-highlight if already highlighted)

`$hl clearAll` – Clear all highlighted entries

`$h1 caseSensitive [1|0]` - Set highlight matching to case sensitive or case insensitive (default is case insensitive)

This release also includes some minor bug fixes.

1.1712.15003.0

This version includes these updates.

TTD memory queries - You can now query TTD for memory accesses similar to how you query for calls today. This allows you to find all of the reads, writes and execution which access a specific range of memory.

Read and write example: `dx @$cursession.TTD.Memory(startAddress, endAddress, "rw")`

Unique execution example: `dx @$cursession.TTD.Memory(startAddress, endAddress, "ec")`

Settings changes - WinDbg Preview will now automatically save settings between sessions, including your symbol path and source path.

JavaScript Improvements

- 64-bit numbers and numerics in JavaScript now contain a modulo method allowing a true 64-bit modulo operation.
- Objects defined in JavaScript can now implement a custom comparable or equatable notion which will work in dx using standard C++ operators or in LINQ operations. In order to utilize this, the script must declare in the initializeScript array that it supports a new version of the host API by inserting a record “new host.apiVersionSupport(1, 2)”. After you’ve done that you can use these functions in any ‘dx’ or Data Model Window LINQ query. If the method implements .compareTo(other), it is comparable (comparison operators work in dx and LINQ). If the method returns a negative value, such as “this < other”. If the method returns zero, “this == other”. If the method returns a positive value “this > other”. If the method implements .equals(other), it is equatable (== works in dx and LINQ). The method must return either true or false.

Minor changes and bug fixes:

- Fixed a bug where the stack and locals windows weren’t working during boot debugging
- Updated the output of LM to more accurately report ProductVersion and similar fields

- Enabled the “step out back” button during TTD sessions
- Added support for -lsrcpath
- The headers in the locals, watch, and model windows now don’t disappear when scrolling down
- When ALT+Tabbing back to WinDbg Preview, the command window will properly preserve cursor location
- Added CTRL+ALT+V shortcut for toggling verbose mode
- You can now disable auto-scrolling of the command window by selecting and holding (or right-clicking) the command window tab and choosing “turn off auto scrolling”
- You can now debug child processes through the launch executable advanced page.

1.0.14.0

This version includes these updates.

Improved process server experience - A new notification in the File menu to show what process server you’re connected to and interacting with has been added. As part of these changes, when ending a debugging session, the process server connection will persist and can be disconnected in the File menu.

New pre-set layout options in the View ribbon - There is a new “Layouts” option in the “View” ribbon. There are currently three layouts: the default, one focused on disassembly, and one minimal.

Time Travel Debugging ribbon - There is an enhanced Time Travel ribbon that will show up when debugging a time travel debugging trace.

Metadata from JavaScript scripts - JavaScript extensions can now return metadata for properties and other constructs. This means that the extension can provide help strings, indicate the display radix for values, and more. Metadata is provided by placing a metadata descriptor on an object via either presence of `Symbol.metadataDescriptor` or an explicit call to `host.metadata.defineMetadata`. Function returns, iterated values, and other value contexts can return metadata for their value via `host.metadata.valueWithMetadata`.

JavaScript API updates - Some potentially source level breaking changes were made to the APIs within the JavaScript provider (including new projected methods and properties on native objects). Existing extensions will not see any of the potentially breaking changes without indicating that they support a new version of the JsProvider API. Support for the new API version is indicated by placing a `host.apiVersionSupport` record

in the array returned by initializeScript with a claim of supporting version 1.1. maybe? .. with a value indicating support for version 1.1.

Changes in API version 1.1 include:

- host.getModuleSymbol and host.getModuleType return null if they cannot find the symbol instead of throwing an exception.
- All native objects have the address property on them in addition to .targetLocation. If the object does not have an address, an exception will be thrown when accessing the property.
- All native objects have new .getObjectValue and .setObjectValue methods on them to access properties on the object which may conflict with names JavaScript places on the object (e.g.: 'address') .

Additional JavaScript changes

- JavaScript extensions can now add and remove properties on data model objects via Object.defineProperty and the delete operator. Adding or registering a JavaScript class as a parent model or type signature is still the strongly preferred way of manipulating the object model.
- JavaScript extensions can now modify global variables within modules in the debug target through a new host.setModuleSymbol API.
- All of the math functions which are on the 64-bit library type (e.g.: .add, .subtract, .multiply, .divide, etc...) are now present on JavaScript numbers as well.
- JavaScript functions and properties can now return values which are enums through custom marshaling. A function or property accessor can return host.typeSystem.marshalAs(value, type...) in order to evoke such custom marshaling.
- The breakpoint command in the script debugger can now break on function names in addition to line/column positions.
- Type objects in JavaScript extensions have access to their containing module through the .containingModule property.

Minor changes and bug fixes:

- Fixed formatting of conditional ribbon tabs to be less confusing.
- Re-worked DML to be stricter in parsing to improve performance.
- Various fixes with the performance and behavior of CTRL+F.
- Added a warning when running un-elevated prior to trying to use TTD.
- Added the option to override automatic target bitness detection.
- Disabled various file menu and ribbon options when they can't be used (like "Go" when in a dump file).

Known issues:

- SOS will not work on x86 traces.

1.0.13.0

This version adds Time Travel Tracing. Time Travel Debugging, allows you to record a process, then replay it later both forwards and backwards. Time Travel Debugging (TTD) can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug. For more information, see [Time Travel Debugging - Overview](#).

1.0.12.0

This version was the first release of WinDbg Preview. For general information on the features available in WinDbg Preview, [Major Features of WinDbg](#).

See Also

[WinDbg – Installation](#)

[WinDbg – Command line startup options](#)

Feedback

Was this page helpful?

 Yes

 No

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

WinDbg - Command line startup options

Article • 10/25/2023



Starting WinDbg

After WinDbg is installed, WinDbgX.exe is available to run from any directory location.

Command line startup options

dbgsyntax

WinDbgX [options]

This following tables summarizes the available command line options.

General Options

Option	Description
-c "command"	Executes a command line after the debugger is attached. This command must be enclosed in quotation marks. Multiple commands can be separated with semicolons.
-v	Enables verbose output in the debugger.
-T <i>Title</i>	Sets the window title.
-logo <i>LogFile</i>	Log Open. Begins logging information to a log file. If the file exists, it will be overwritten.
-loga <i>LogFile</i>	Log Append. Begins logging information to a log file. If the file exists, it will be appended to.
-e <i>EventHandle</i>	Signals the event with the given handle after the next exception in a target.
-?	Displays a summary of commands available.

Kernel Options

Option	Description
-k [ConnectType]	Starts a kernel debugging session. If -k is used without any <i>ConnectType</i> options following it, it must be the final entry on the command line.
-kqm	Starts KD in quiet mode.
-kl	Starts a kernel debugging session on the same machine as the debugger.
-kx <i>ExdiOptions</i>	Starts a kernel debugging session using an EXDI driver. For more information about EXDI, see Configuring the EXDI Debugger Transport .
-d	After a reboot, the debugger will break into the target computer as soon as a kernel module is loaded.

User Mode Options

Option	Description
-o	Debugs all processes launched by the target application (child processes).
-g	Ignores the initial breakpoint in target application.
-G	Ignores the final breakpoint in target application.
-pv	Specifies that the debugger should attach to the target process noninvasively.
-hd	Specifies that the debug heap should not be used.
-cimp	Specifies that any processes created will use an implicit command-line set by the server instead of a user-given command-line string from the client.

Target Options

Option	Description
-remote <i>ClientTransport</i>	Connects to a debugging server that is already running. For an explanation of the possible <i>ClientTransport</i> values, see Activating a Debugging Client . When this parameter is used, it must be the first parameters on the command line.
-server <i>ServerTransport</i>	Creates a debugging server that can be accessed by other debuggers. For an explanation of the possible <i>ServerTransport</i> values, see Activating a Debugging Server .
-premote <i>SmartClientTransport</i>	Creates a smart client, and connects to a process server that is already running. For an explanation of the possible <i>SmartClientTransport</i> values, see Activating a Smart Client .

Option	Description
<code>-p PID</code>	Specifies the decimal process ID to be debugged.
<code>-tid TID</code>	Specifies the thread ID of a thread to be resumed when the debugging session is started.
<code>-psn ServiceName</code>	Specifies the name of the service contained in the process to be debugged. This is used to debug a process that is already running.
<code>-pn ProcessName</code>	Specifies the name of the process to be debugged.
<code>-z DumpFile</code>	Specifies the name of a crash dump file to debug. If the path and file name contain spaces, this must be surrounded by quotation marks.
<code>-debugArch x86 -or- amd64</code>	Override the autodetect behavior and set the target bitness for the debugger.
<code>-loadSession</code>	Load a saved session configuration file.
<code>-setupFirewallRules</code>	Configures the required firewall rules on the local system to allow kernel debugging using KDNET.
<code>-openPrivateDumpByHandle Handle</code>	<i>Microsoft internal use only.</i> Specifies the handle of a crash dump file to debug.
<code>-benchmarkStartup</code>	<i>Microsoft internal use only.</i> Runs a startup benchmark and appends the result to a file.

Symbol Options

Option	Description
<code>-y SymbolPath</code>	Specifies the symbol path to use. Separate multiple paths with a semicolon (;). If the path contains spaces, it should be enclosed in quotation marks. For details, and for other ways to change this path, see Symbol Path .
<code>-n</code>	Noisy symbol load. Enables verbose output from symbol handler.
<code>-i ImagePath</code>	Sets the image search path to use.
<code>-sdce</code>	Causes the debugger to display 'File access error' messages during symbol load.
<code>-ses</code>	Causes the debugger to perform a strict evaluation of all symbol files and ignore any questionable symbols.
<code>-sicv</code>	Causes the symbol handler to ignore the CV record

Option	Description
-sins	Causes the debugger to ignore the symbol path and executable image path environment variables.
-snc	Causes the debugger to turn off C++ translation.
-snul	Disables automatic symbol loading for unqualified names.
-sup	Causes the symbol handler to search the public symbol table during every symbol search
-sflags	Sets all the symbol handler options at once.

Source Path Options

Option	Description
-srcpath	Specifies the source path to use on the debugging server.
-lsrcpath	Specifies the source path to use on the local client.

If you are in a local debugger session, srcpath and lsrcpath are effectively the same (your “server” is your local session). For remote debugging there are situations where you may want to set these to different values. For more information about remote debugging see, [Remote Debugging](#).

Exception handling

Option	Description
-x	Enable second-chance handling only for access violation exceptions.
-xe <i>Exception</i>	Enable first-chance exception handling for the specified exception.
-xd <i>Exception</i>	Enable second-chance exception handling for the specified exception.
-xn <i>Exception</i>	For the given exception, disable first- and second-chance-handling, and only display a message on the console.
-xi <i>Exception</i>	Completely ignore the given exception, disabling first- and second-chance handling, and not outputting anything to the console.

For a list of exceptions that can be specified, see [Event Definitions and Defaults](#).

Post Mortem

Option	Description
-I	Sets WinDbg as the default post-mortem debugger for the system.
-IS	Sets WinDbg as the default post-mortem debugger for the system silently, with only errors being reported.

Deprecated Options

Option	Description
-Q	Deprecated command-line option.
-QY	Deprecated command-line option.
-QS	Deprecated command-line option.
-QSY	Deprecated command-line option.
-WX	Deprecated command-line option.

For general information on the startup parameters, see [WinDbg Command-Line Options](#).

You can use -? to list the supported command line options.



See Also

[WinDbg Features](#)

WinDbg - Settings and workspaces

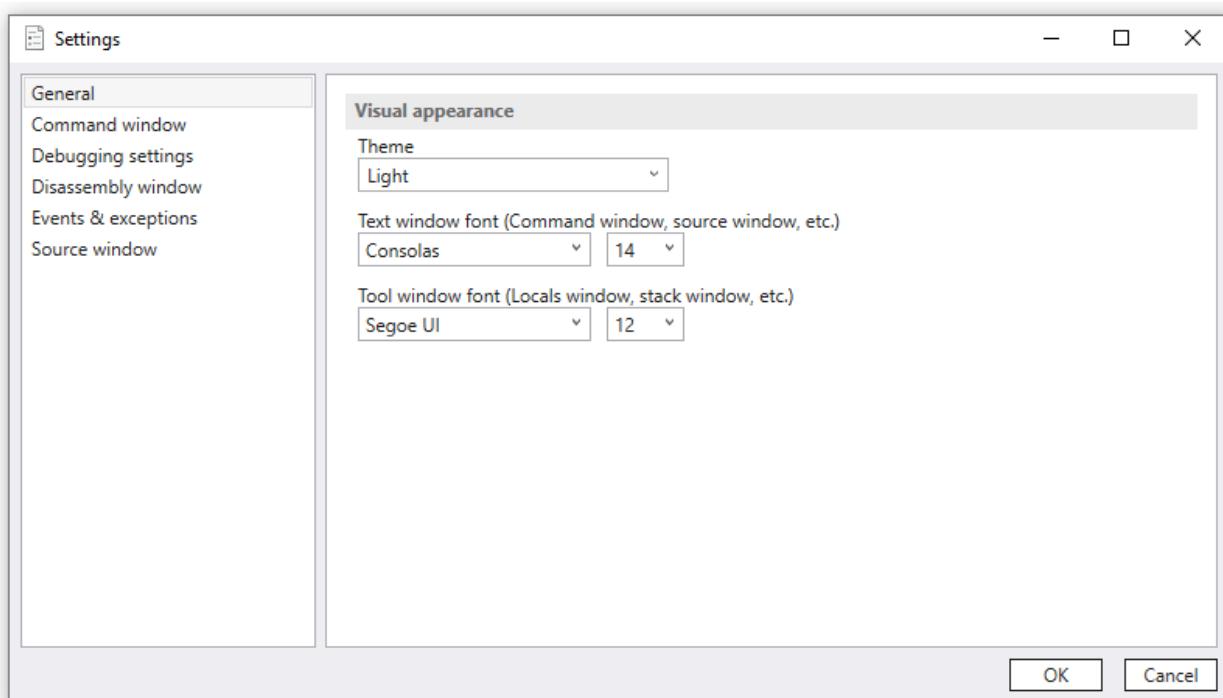
Article • 10/25/2023



This section describes how to setup and configure the WinDbg debugger.

Settings

Use the settings menu to set things such as the source and symbol path as well as choose the light and dark theme for the debugger.



There are currently six settings dialog panels:

- General
- Command Window
- Debugging Settings
- Disassembly windows
- Events & exceptions
- Source window

For more information on setting the paths, see [Symbol path for Windows debugger](#) and [Source Code Debugging in WinDbg \(Classic\)](#).

Workspaces

Workspaces allows you to save configuration information in the target connection information file.

The options in workspaces are saved upon closing the debugger or can be manually saved using *File -> Save Workspace*.

Workspaces are automatically loaded when launching from the recent targets list or they can be manually loaded in the file menu.

In addition to the target connection information, the following settings are stored in the workspaces file.

General Settings

ⓘ Note

This list and format isn't final and is subject to change.

Setting	Default	Description
FinalBreak	true	If true, ignores the final breakpoint (-g command-line option).
SourceDebugging	true	Toggles between source or assembly mode.
DebugChildProcesses	false	(User mode only) If true will debug child processes launched by the target application. (-o command-line option).
Noninvasive	false	Specifies non-invasive attach (-pv command-line option).
NoDebugHeap	false	Specifies the debug heap should not be used (-hd command-line option).
Verbose	false	When verbose mode is turned on, some display commands (such as register dumping) produce more detailed output. (-v command-line option).
Elevate	-	Used internally by WinDbg - Do not modify.
Restartable	-	Used internally by WinDbg - Do not modify.
UseImplicitCommandLine	false	Use implicit command-line (-cimp command-line option). This starts the debugger with an implicit command line

Setting	Default	Description
		instead of an explicit process to run.

For more information about the command line options, see [WinDbg Command-Line Options](#).

Symbol Settings

Setting	Default	Description
SymbolOptionsOverride	0	An explicit symbol option mask, in the form of a single hex number.
ShouldOverrideSymbolOptions	false	If set to <i>true</i> override all of the symbol options listed below with the provided symbol option mask, described above.
SymOptExactSymbols	false	This option causes the debugger to perform a strict evaluation of all symbol files.
SymOptFailCriticalErrors	false	This symbol option causes file access error dialog boxes to be suppressed.
SymOptIgnoreCvRec	false	This option causes the symbol handler to ignore the CV record in the loaded image header when searching for symbols.
SymOptIgnoreNtSympath	false	This option causes the debugger to ignore the environment variable settings for the symbol path and the executable image path.
SymOptNoCpp	false	This symbol option turns off C++ translation. When this symbol option is set, :: is replaced by __ in all symbols.
SymOptNoUnqualifiedLoads	false	This symbol option disables the symbol handler's automatic loading of modules. When this option is set and the debugger attempts to match a symbol, it will only search modules which have already been loaded.
SymOptAutoPublics	false	This symbol option causes DbgHelp to search the public symbol table in a .pdb file only as a last resort. If any matches are found when searching the private symbol data, the public symbols will not be searched. This improves symbol search speed.
SymOptDebug	false	This symbol option turns on noisy symbol loading. This instructs the debugger to display information

Setting	Default	Description
		about its search for symbols.

For more information on symbol options, see [Symbol Options](#).

Window layout settings

Window layout is saved globally and are not saved in the workspaces file.

Workspaces XML file

The workspace and target connection information is stored in XML format.

The following file, shows an example workspaces configuration file.

```
XML
```

```
<?xml version="1.0" encoding="utf-8"?>
<TargetConfig Name="C:\paint.dmp" LastUsed="2017-08-03T21:34:20.1013837Z">
  <EngineConfig />
  <EngineOptions>
    <Property name="FinalBreak" value="true" />
    <Property name="SourceDebugging" value="true" />
    <Property name="DebugChildProcesses" value="false" />
    <Property name="Noninvasive" value="false" />
    <Property name="NoDebugHeap" value="false" />
    <Property name="Verbose" value="false" />
    <Property name="SymbolOptionsOverride" value="0" />
    <Property name="ShouldOverrideSymbolOptions" value="false" />
    <Property name="SymOptExactSymbols" value="false" />
    <Property name="SymOptFailCriticalErrors" value="false" />
    <Property name="SymOptIgnoreCvRec" value="false" />
    <Property name="SymOptIgnoreNtSympath" value="false" />
    <Property name="SymOptNoCpp" value="false" />
    <Property name="SymOptNoUnqualifiedLoads" value="false" />
    <Property name="SymOptAutoPublics" value="false" />
    <Property name="SymOptDebug" value="false" />
    <Property name="Elevate" value="false" />
    <Property name="Restartable" value="true" />
    <Property name="UseImplicitCommandLine" value="false" />
  </EngineOptions>
  <TargetOptions>
    <Option name="OpenDump">
      <Property name="DumpPath" value="C:\paint.dmp" />
    </Option>
  </TargetOptions>
</TargetConfig>
```

Note that this file format continues to evolve as more features are added to the WinDbg debugger.

See Also

[WinDbg Features](#)

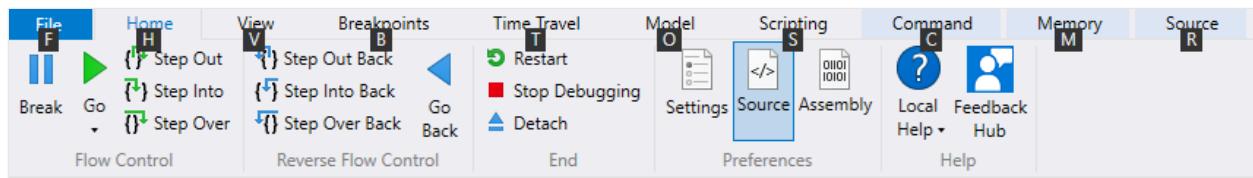
WinDbg keyboard shortcuts

Article • 11/15/2024



This section summarizes the keyboard shortcuts for the WinDbg debugger.

All of the ribbon menu options are available using the **Alt** + the first letter of the option. For example **Alt** + **H** to go to the home menu, **Alt** + **H** + **S** to stop debugging.



Flow control

[Expand table](#)

Keystroke	Description
F5	Continue
F10	Step over
F11	Step Into
Shift+F11	Step out
F7	Run to line
Ctrl+Shift+I	Set instruction pointer to highlighted line
Ctrl+Break or Alt+Del	Break
Ctrl+Shift+F5	Restart
Shift+F5	Stop debugging
Alt+H,D	Detach

Setup

[\[\] Expand table](#)

Keystroke	Description
F6	Attach to process
Ctrl+R	Connect to remote
Ctrl+D	Open dump file
Ctrl+K	Attach to kernel debugger
Ctrl+E	Launch process
Ctrl+P	Launch app package

Breakpoints

[\[\] Expand table](#)

Keystroke	Description
F9	Toggle breakpoint on highlighted line
Ctrl + F9	Toggle breakpoint enabled state on highlighted line
Shift + F9	Add breakpoint
Ctrl+Alt+K	Toggle initial break

Windowing

[\[\] Expand table](#)

Keystroke	Description
Ctrl+Tab	Open window changer
Alt+1	Open/focus on command window
Alt+2	Open/focus on watch window
Alt+3	Open/focus on locals window
Alt+4	Open/focus on registers window
Alt+5	Open/focus on memory window

Keystroke	Description
Alt+6	Open/focus on stack window
Alt+7	Open/focus on disassembly window
Alt+8	Open/focus on breakpoints window
Alt+9	Open/focus on thread window

Scripting

[\[+\] Expand table](#)

Keystroke	Description
Ctrl+Shift+O	Open script
Ctrl+Shift+Enter	Execute script
Ctrl+S	Save script
Alt+S,N	New script
Alt+S,U	Unlink script

Stack navigation

[\[+\] Expand table](#)

Keystroke	Description
Ctrl+↑ / ↓	Move up/down a stack frame

Help

[\[+\] Expand table](#)

Keystroke	Description
F1	Open help file
Shift+F1	Search selection online (source window)

Misc.

Expand table

Keystroke	Description
Ctrl+Alt+V	Toggle Verbose Mode

See Also

[WinDbg Features](#)

Feedback

Was this page helpful?

 Yes

 No

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

WinDbg - Start a user mode session

Article • 10/25/2023



This section describes how to start a user mode session with the WinDbg debugger.

Select *File, Start debugging*, and select either of these four options:

- *Launch Executable* - Starts an executable and attaches to it by browsing for the target.
- *Launch Executable (advanced)* - Starts an executable and attaches to it using a set of dialog boxes with advanced options.
- *Attach to a process* - Attaches to an existing process.
- *Launch App Package* - Launches and attaches to an app package.

All four options are described here.

Launch Executable

Use this option to starts an executable and attach to it.

Browse to the desired executable in the provided file dialog and open it.

Launch Executable (advanced)

Use this option to start an executable and attach to it using a set of text boxes with advanced options.

Specify the following options:

- Path to the executable, such as C:\Windows\notepad.exe
- Optional arguments to provide to the executable when launched
- Optional start directory location
- Target bitness, auto 32 or 64.
- Enable Debug child process
- Record with Time Travel Debugging

Start debugging

The screenshot shows the 'Start debugging' dialog. On the left, a sidebar lists several options: 'Recent' (disabled), 'Launch executable', 'Launch executable (advanced)' (selected), 'Attach to process' (disabled), 'Open dump file' (disabled), 'Open trace file' (disabled), and 'Connect to remote debugger' (disabled). The main area contains fields for 'Executable' (C:\DisplayGreeting\Debug\DisplayGreeting.exe), 'Arguments' (Hello), 'Start directory' (C:\MyData), 'Target bitness' (Autodetect), and a checked 'Debug child processes' checkbox. Below these are two informational messages: 'WinDbg needs to be run elevated to use Time Travel Debugging' and 'Record with Time Travel Debugging'. A large 'OK' button is at the bottom right.

Attach to a process

Use this option to attach to an existing process.

Select *Show process from all users* to show additional processes.

Use the search box to filter down the process list, for example by searching for SystemApps.

ⓘ Note

Items with a UAC shield will need the debugger to be elevated to attach.

Use the pull down dialog on the *Attach* button to select *non-invasive attach*.

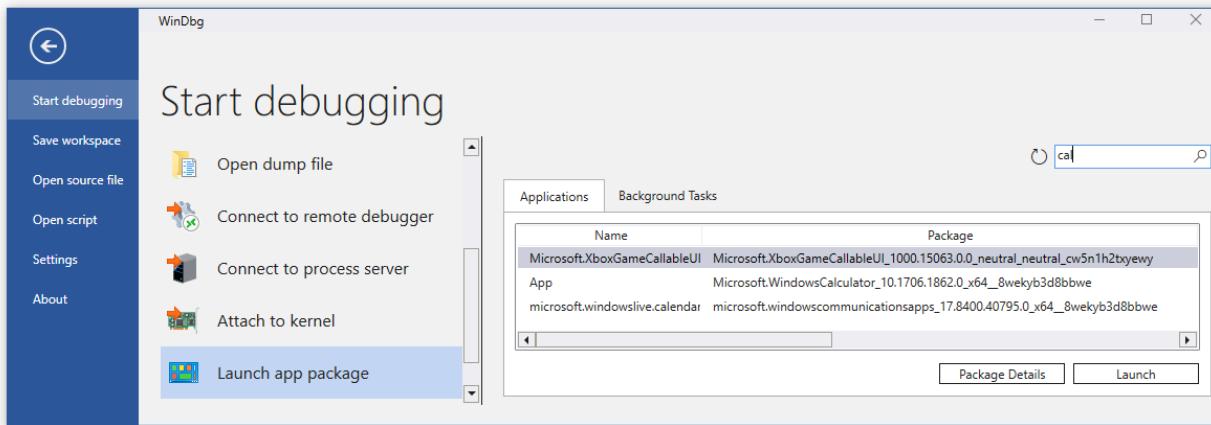
The screenshot shows the 'Start debugging' dialog with the 'Attach to process' option selected in the sidebar. The main area displays a table of running processes:

Process	PID	Platform	User	Session	Command line	Services
MicrosoftEdge.exe	21252	64 bit	REDMOND\	1	"C:\Windows\SystemApps\MicrosoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe"	
ShellExperienceHost.exe	2880	64 bit	REDMOND\	1	"C:\Windows\SystemApps\ShellExperienceHost_8wekyb3d8bbwe\ShellExperienceHost.exe"	
LockApp.exe	6628	64 bit	REDMOND\	1	"C:\Windows\SystemApps\Microsoft.LockApp_8wekyb3d8bbwe\LockApp.exe"	
TextInputHost.exe	12292	64 bit	REDMOND\	1	"C:\Windows\SystemApps\Microsoft.TextInputHost_8wekyb3d8bbwe\TextInputHost.exe"	
SearchApp.exe	11268	64 bit	REDMOND\	1	"C:\Windows\SystemApps\Microsoft.Search_8wekyb3d8bbwe\SearchApp.exe"	
StartMenuExperienceHost	10992	64 bit	REDMOND\	1	"C:\Windows\SystemApps\Microsoft.StartMenuExperienceHost_8wekyb3d8bbwe\StartMenuExperienceHost.exe"	

Below the table are checkboxes for 'Show processes from all users' and 'Target bitness: Autodetect'. At the bottom right is a dropdown menu with 'Attach' and a downward arrow.

Launch App Package

Use this option to launch and attach to an app package using either the Applications or Background Task tabs. Use the search box to locate a specific app or background task. Use the Package Details button to display information about the package.



See Also

[WinDbg Features](#)

WinDbg - Starting a kernel mode session

Article • 10/25/2023

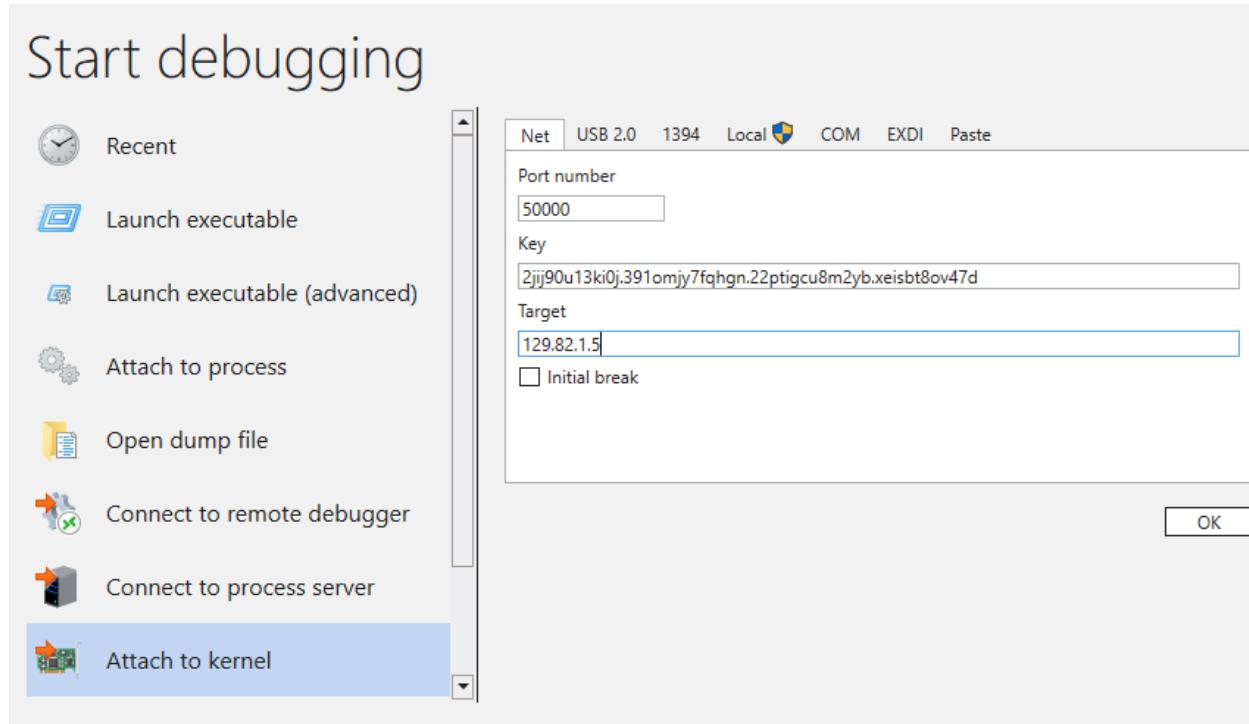


This topic describes how to start a kernel mode session with WinDbg.

The process is very similar to how it has been done with previous versions of WinDbg. Select the tab for the type of transport you're using, fill in the required fields, and click OK.

ⓘ Note

Local kernel debugging requires WinDbg to be launched elevated.



The *Paste* tab allows you to paste in custom connection strings.

If you are not familiar with setting up a debugger kernel mode session, see [Getting Started with WinDbg \(Kernel-Mode\)](#)

See Also

WinDbg Features

WinDbg - Start a remote, process server and dump file session

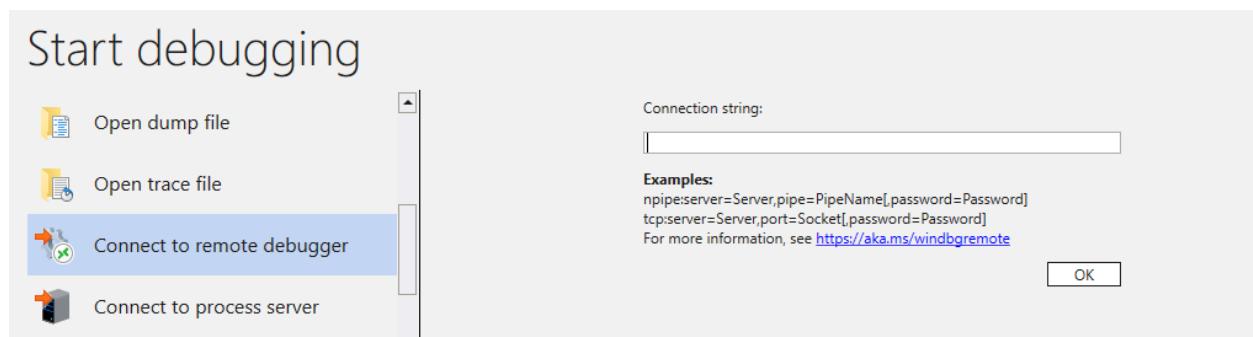
Article • 10/25/2023



This section describes how to start a remote, process server and dump file session with the WinDbg debugger.

Remote Debug Server

Use this option to connect to a remote debugging server.



Remote debugging involves two debuggers running at two different locations. The debugger that performs the debugging is called the debugging server. The second debugger, called the debugging client, controls the debugging session from a remote location. To establish a remote session, you must set up the debugging server first and then connect to it with the debugging client.

For more information about remote sessions, see [Remote WinDbg Features](#).

Process Debug Server

Remote debugging through a process server involves running a small application called a process server on the server computer. Then a user-mode debugger is started on the client computer. Since this debugger will be doing all of the actual processing, it is called the smart client.

For more information about process server sessions, see [Process Servers \(User Mode\)](#).

Open a dump file

To open a dump file, browse to the desired file in the provided file dialog and open it.

For more information about the different types of dump files, see [Analyze crash dump files by using WinDbg](#).

See Also

[WinDbg Features](#)

WinDbg - File Menu

Article • 10/25/2023



This topic describes how to use the file menu.

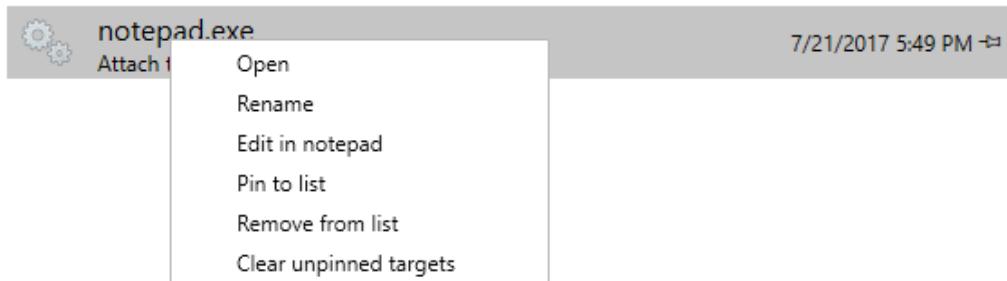
Start debugging

When you first open the file menu, you'll see *Start debugging* and your recent debugger targets. Use *Start debugging* to configure new and open previous debugger sessions.

Recent

The recent list contains a list of your recent workspaces and debugger connections. For more information on working settings and workspaces see [WinDbg Setup – Settings and workspaces](#).

You can use the right click menu to manage your workspaces, like pinning, renaming and moving them. As well as editing them in notepad.



Start a new session

Use the other tabs in the *Start debugging* section to start a new debugger session, like attaching or launching a process. For more information on starting a new session see [WinDbg - Start a user-mode session](#) and [WinDbg - Start a kernel mode session](#)

Save workspace

Use *Save workspace* to save the current workspace.

Session connection information is stored in workspace configuration files. Workspace files are stored with a .debugTarget file extension.

The default location for workspace files is:

Console

C:\Users*UserName*\AppData\Local\DBG\targets

Open source file

Use *Open source file* to open a source file. Do this when you want to work with additional source files that have not been loaded because of code execution. For more information on working with source files, see [Source Code Debugging in WinDbg \(Classic\)](#)

Open script

Use *Open script* to open an existing Javascript or NatVis script. For more information on working with scripts see [WinDbg - Scripting Menu](#).

Settings

Use the settings menu to set the source and symbol path as well as choose the light and dark theme for the debugger. For more information on settings see [WinDbg Setup – settings and workspaces](#).

About

Use *About* to display build version information for the debugger. You can also use this screen to view the Microsoft privacy statement.

Exit

Use *Exit* to exit the debugger.

See Also

[WinDbg Features](#)

WinDbg - Home Menu

Article • 10/25/2023



This topic describes how to work with the home menu.



Flow Control

Use the *Flow Control* buttons to break into a connected debugging target, resume code execution on the target and step in to and out of code.

Reverse Flow Control

Use the *Reverse Flow Control* buttons to travel back in time. For more information, see [Time Travel Debugging - Overview](#).

End

Use the *End* buttons to restart, detach and stop debugging.

Preferences

Use *Preferences* buttons to toggle between source code and assembly views and to access the *Settings* menu.

Help (Support)

Use *Help* buttons to do the following:

- Review Local Help - Offline
- Online Help - Most up to date

GitHub Feedback

- If you find any bugs or have a feature request, you can follow the feedback button in the ribbon to go to the [GitHub page ↗](#) where you can file a new issue.

See Also

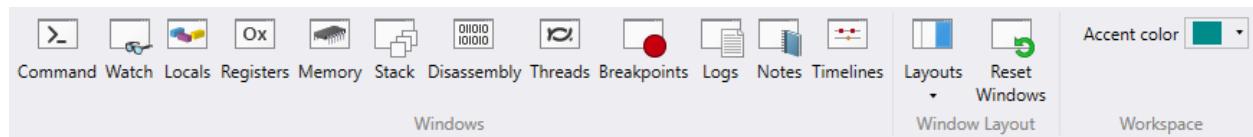
[WinDbg Features](#)

WinDbg - View Menu

Article • 10/25/2023



This section describes how work with the view menu in WinDbg.



The view menu will open a new Window for each item, or bring focus to the existing Window, if one is already open.

Command

The command Window allows you to enter debugger commands. For more information about debugger commands, see [Debugger Commands](#).

Watch

The watch Window allows you to watch local variables and registers.

The locals and watch windows are both based off of the data model that is used by the dx command. This means the locals and watch windows will benefit from any NatVis or JavaScript extensions you have loaded, and support full LINQ queries, just like the dx command. For more information about the data model, see [WinDbg - Data Model](#).

Locals

The locals window displays information about all of the local variables in the current scope. The locals window will highlight values that have changed during the previous code execution.

Locals			
Name	Value	Type	
AddDeviceFunction	0xffff80a2a781000	long (<code>_cdecl</code>)(_DRIVER_OBJECT * _)	
DevObjType	VF_DEVOBJ_FDO (4)	VF_DEVOBJ_TYPE	
DriverObject	0xffffa48f66012170 : Driver "\Driver\sysvad_tabletaudio"	_DRIVER_OBJECT *	
[<Raw View>]	Driver "\Driver\sysvad_tabletaudiosample"	_DRIVER_OBJECT	
HardwareDatabase	0xffff80252843db8 : "\REGISTRY\MACHINE\HARDWAF	_UNICODE_STRING *	
DeviceObject	0xffffa48f68a06060 : Device for "\Driver\sysvad_tabletau	_DEVICE_OBJECT *	
Flags	0x12	bool	
LegacyDriver	true		
UnknownFlags	0x10		
Devices			
PhysicalDeviceObject	0xffffa48f609fbe40 : Device for "\Driver\PnpManager"	_DEVICE_OBJECT *	
Locals			
Watch			

Registers

Registers displays the contents of the processors registers when they are available. For more information about registers, see [Registers](#) and [Viewing and Editing Registers in WinDbg \(Classic\)](#).

Memory

Use the memory window to display memory locations. In addition to providing a memory address, you can use the Pseudo-Register values such as \$scopeip and \$eventip to examine memory. Pre-append the @ symbol to use the pseudo-register values in the memory window, for example, @\$scopeip. For more information, see [Pseudo-Register Syntax](#)

Stack

Use the stack Window to view the current call stack. The stack window provides basic highlighting of the current frame.

Disassembly

The disassembly window highlights the current instruction and retains that position when you scroll.

```
Address: @@$scopeip
0012167e cc int 3
0012167f cc int 3
DisplayGreeting!main:
00121680 55 push ebp
00121681 8bec mov ebp, esp
00121683 81eca4000000 sub esp, 0A4h
00121689 53 push ebx
0012168a 56 push esi
0012168b 57 push edi
0012168c 6a64 push 64h
0012168e 6a00 push 0
00121690 8d459c lea eax, [ebp-64h]
00121693 50 push eax
00121694 e802faffff call DisplayGreeting!ILT+150(_memset) (0012109b)
00121699 83c40c add esp, 0Ch
0012169c 6a64 push 64h
0012169e 8d4d9c lea ecx, [ebp-64h]
001216a1 e89afaffff call DisplayGreeting!ILT+315(?data?$array_W$0DCstdQAEPA_WXZ) (0012109c)
001216a6 50 push eax
```

Threads

The threads window highlights the current thread.

Breakpoints

Use the breakpoints window to view, enable and clear breakpoints.

Location	Line	Type	Hit Count	Function
c:\data1\redstone\wdk driver samples - april 20	1333	Software	1	tableaudiosample!CMiniportWaveF
c:\data1\redstone\wdk driver samples - april 20	550	Software	1	tableaudiosample!AddDevice
c:\data1\redstone\wdk driver samples - april 20	581	Software	1	tableaudiosample!AddDevice+0x9c

Logs

This log is of the WinDbg internals. It can be viewed to monitor long running processes and for troubleshooting the debugger itself.

You can continue to create a traditional debugger command log, using the .logopen command. For more information on that, see [Keeping a Log File in WinDbg](#).

Notes

Use the Notes option to open a note taking window.

Timelines

Use Timelines to open or bring focus to the timelines window. For more information on timelines, see [WinDbg - Timelines](#).

Modules

Use modules to display loaded modules and their related information. Modules displays the following:

- The name of the module including the path location
- The size in bytes of the loaded module
- The base address that the module is loaded at
- The file version

Modules	X			
Name	Size	Base Address	File Version	
E:\Data1\Vibranium\Debugger\Timeline\Disp...	0x1e000	0x50000		
C:\WINDOWS\SYSTEM32\apphelp.dll	0x9f000	0x6b4a0000	10.0.18362.1	
C:\WINDOWS\System32\KERNEL32.DLL	0xe0000	0x74af0000	10.0.18362.10022	
C:\WINDOWS\System32\KERNELBASE.dll	0x1fc000	0x756e0000	10.0.18362.10024	
C:\WINDOWS\SYSTEM32\ntdll.dll	0x19a000	0x76fd0000	10.0.18362.10024	

Layouts

Use the Layouts pull down menu to select from three window layouts.

Reset Windows

Use this function to reset the debugger windows to their default positions.

Accent Color

Use the pull down menu to set the accent color for the debugger.

See Also

[WinDbg Features](#)

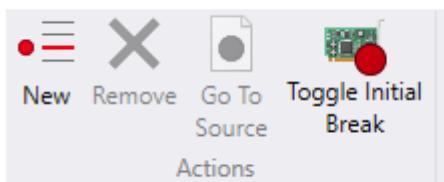
WinDbg - Breakpoints Menu

Article • 10/25/2023

This section describes how to work with breakpoints using the WinDbg debugger.

Breakpoints Menu

Use the breakpoints menu to create new and remove existing breakpoints as well as toggle the initial breakpoint (initial breakpoint is currently kernel only).



Breakpoints Window

Use the breakpoints window, opened via the View menu, to view, set and clear breakpoints. You can also double-click a breakpoint to open its source file.

Breakpoints					
Location	Line	Type	Hit Count	Function	
✓ c:\data1\redstone\wdk driver samples - april 20	1333	Software	1	tabletaudiosample\CMiniportWavef	
✓ c:\data1\redstone\wdk driver samples - april 20	550	Software	1	tabletaudiosample\AddDevice	
✓ c:\data1\redstone\wdk driver samples - april 20	581	Software	1	tabletaudiosample\AddDevice+0x9c	

The breakpoint window keep a running total of each time the breakpoint is hit.

The general process of working with breakpoints is similar to previous versions of WinDbg. For more information about setting breakpoints, see [Setting Breakpoints in WinDbg \(Classic\)](#).

WinDbg - Time Travel Menu

Article • 10/25/2023



This section describes how work with the time travel menu in WinDbg.



For more information about time travel see [Time Travel Debugging - Overview](#).

Trace

Index Trace

Use the Index Trace option to force a re-index of a time travel trace.

Events

Use the Events pull down to display either *Exceptions* or *Module load* events.

Position

Time travel to start

Use Time travel to start to move to the start of a time travel trace file.

Time travel to end

Use Time travel to start to move to the end of a time travel trace file.

Misc

Information

Use Information to display information about the trace, such as size and number of threads.

Timelines

Use the `to Timelines` option to access debugger timelines. For more information see [WinDbg Timelines](#).

See Also

[WinDbg Features](#)

WinDbg - Data Model Menu

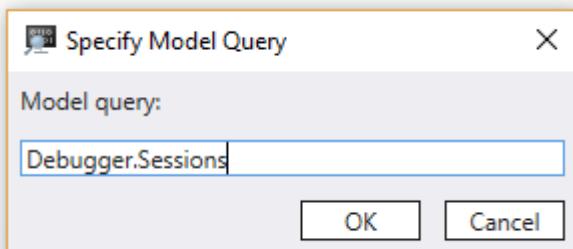
Article • 10/25/2023

This section describes how to work with the data model menu in the WinDbg debugger.

New Model Query

Use the New Model Query dialog to create a new model query. You can put anything here you'd put into a normal `dx` command.

For example, specify `Debugger.Sessions` to examine the debugger sessions objects.



For general information about the debugger objects refer to [dx \(Display Debugger Object Model Expression\)](#).

Use LINQ queries to dig deeper into the session. This query shows the top 5 processes running the most threads.

```
dbgcmd

Debugger.Sessions.First().Processes.Select(p => new { Name = p.Name,
ThreadCount = p.Threads.Count() }).OrderByDescending(p => p.ThreadCount),5
```

Name	Value	Type
[0x4]		
Name	"<Unknown Image>"	
ThreadCount	0x000000000000000b7	
[0x61c]		
Name	"explorer.exe"	
ThreadCount	0x0000000000000004f	
[0xfc]		
Name	"Music.UI.exe"	
ThreadCount	0x00000000000000045	
[0xa10]		
Name	"RuntimeBroker.exe"	
ThreadCount	0x0000000000000003c	
[0xa7c]		
Name	"ShellExperienceHost.exe"	
ThreadCount	0x0000000000000002e	
[...]		

Data Model Explorer

Use the data model explorer to quickly browse every data model object in the `Debugger` namespace.

Name	Value	Type
Sessions		
[0x0]	Live user mode: <Local>	
Settings		
State		
DebuggerVariables		
cursession	Live user mode: <Local>	
Processes		
[0x676c]	notepad.exe	BSTR
Name	notepad.exe	
Id	0x676c	unsigned __int...
Handle	0x2d8	unsigned __int...
Threads		

Display Mode

Use display mode to toggle between grid and hierarchy display mode. You can right-click column headers to hide or show more columns.

Grid mode can be useful to dig down in the objects. For example, here is the previous top threads query in grid view.

Name	ThreadCount	[0x384]	[0x177c]	[0xce0]	[0x658]	[0xe70]
[0x4]	0xb7					
[0x61c]	0x4f					
[0xfac]	0x45					
[0x1a10]	0x3c					
[0xa7c]	0x2e					
[...]		[...]				

When you click on any underlined item a new tab is opened and a query is run to display that information.

This query shows the devices in the plug and play device tree grouped by the name of the physical device object's driver for a kernel session.

dbgcmd
Debugger.Sessions.First().Devices.DeviceTree.Flatten(n => n.Children).GroupBy(n => n.PhysicalDeviceObject->Driver->DriverName.ToString())

Debugger.Sessions			
Name	[0x0]	[0x1]	[0x2]
["\\Driver\\PnpManager\\"]	HTREE\\ROOT\\0	ROOT\\volmgr\\0C	ROOT\\sysvad_Tal
["\\Driver\\SoftwareDevice\\"]	SWD\\RADIO\\3DB5895D-CC28-44B3-AD3D-6F	SWD\\MMDEVAP	SWD\\PRINTENUI
["\\Driver\\volmgr\\"]	STORAGE\\Volume\\{59787bf2-1bfc-11e7-84dd-	STORAGE\\Volum	STORAGE\\Volum
["\\Driver\\ACPI HAL\\"]	ACPI_HAL\\PNP0C08\\0 (ACPI)		
["\\Driver\\volsnap\\"]	STORAGE\\VolumeSnapshot\\HarddiskVolumeSr	STORAGE\\Volum	
["\\Driver\\ACPI\\"]	ACPI\\PNP0A08\\2&daba3ff&1 (pci)	ACPI\\PNP0C01\\2	ACPI\\PNP0C0C\\2
["\\Driver\\pci\\"]	PCI\\VEN 8086&DEV D131&SUBSYS 304A103C	PCI\\VEN 8086&I	PCI\\VEN 8086&I
["\\Driver\\usbehci\\"]	USB\\ROOT_HUB20\\4&1a3cc8c0&0 (usbbus)	USB\\ROOT_HUB	
["\\Driver\\HDAudBus\\"]	HDAUDIO\\FUNC 01&VEN 10EC&DEV 0662&S		
["\\Driver\\iaStorA\\"]	SCSI\\Disk&Ven ST350041&Prod ST3500418AS	SCSI\\CdRom&Ve	
["\\Driver\\nvlddmkm\\"]	DISPLAY\\DELF015\\5&3817e4c2&0&UID10488&		
["\\Driver\\usbhub\\"]	USB\\VID 8087&PID 0020\\5&1aaec2e&0&1 (u	USB\\VID 8087&I	USB\\VID 0409&I
["\\Driver\\HidUsb\\"]	HID\\VID 045E&PID 076C&CoI01\\7&f7db0c9&	HID\\VID 045E&F	HID\\VID 045E&F
["\\Driver\\usbccgp\\"]	USB\\VID 045E&PID 0750&MI 00\\9&2c4abd71	USB\\VID 045E&I	USB\\VID 045E&I

Change Query

Use change query to change the query that is used in the active data model window.

See Also

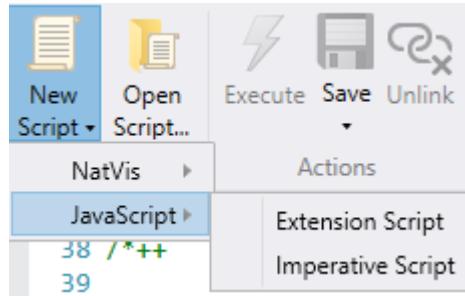
[dx \(Display Debugger Object Model Expression\)](#)

[WinDbg Features](#)

WinDbg - Scripting Menu

Article • 10/25/2023

This section describes how to use the scripting support in the WinDbg.



The WinDbg script window features basic syntax highlighting, IntelliSense, and error recognition.

Use the ribbon buttons to:

- Create a new script
- Open an existing script
- Execute a script
- Save a script
- Unlink a script

You can also automatically execute scripts by right-clicking in the script window and selecting *Execute Script on Save*. When you successfully load a script, a green check box will appear on the script title bar. If there are errors in the script, a red x will be displayed.

JavaScript Scripting

To start using JavaScript, you must first be debugging a target. When you're ready to start working on your JavaScript, click "Load JavaScript Provider". After that you can create a new JavaScript, by picking between these two types of script templates.

- **Extension Script** - A script which is designed to act as an extension to the debugger. It manipulates the object model of the debugger and provides continued functionality. No action happens upon hitting the *Execute* button on the ribbon.
- **Imperative Script** - A script which is designed to perform an action each and every time the *Execute* button is clicked on the ribbon. Such a script does not generally modify the object model of the debugger.

For more information about working with JavaScript, see these topics:

[JavaScript Debugger Scripting](#)

[Native Debugger Objects in JavaScript Extensions](#)

[JavaScript Debugger Example Scripts](#)

NatVis Scripting

Use **New Script > NatVis** to open the following blank NatVis template.

XML

```
<AutoVisualizer
  xmlns="https://schemas.microsoft.com/vstudio/debugger/natvis/2010">
  <Type Name="">
    </Type>
</AutoVisualizer>
```

For more information about working with NatVis, see [Debugger Objects in NatVis](#).

See Also

[WinDbg Features](#)

WinDbg - Notes, Command, Memory and Source Menus

Article • 10/25/2023

This section describes how work with the Notes, Command, Memory and Source menus in WinDbg.

Notes

Use the menu to:

- Open a notes file
- Save a notes file

Command

Use the command menu to:

- Prefer DML
- Highlight and Un-highlight the current text selection (CTRL+ALT+H)
- Clear the command window text
- Save window text to a dml file

Memory

Use the memory menu to:

- Set a data model memory query
- Set the memory size, for example to byte or long
- Set the display format, for example hex or signed
- Set the text display format, for example to ASCII

Source

Use the source menu to:

- Open a source file
- Set an instruction pointer
- Run to cursor

- Close all source windows

See Also

[WinDbg Features](#)

WinDbg - Restricted Mode

Article • 10/25/2023

Restricted Mode



This section describes how to enable the restricted mode feature that restricts the type of debugging sessions that can be started.

WinDbg provides you the ability to start a variety of debugging session types. However, in some circumstances you may not want WinDbg to be able to start certain debugging sessions. Restricted Mode limits the types of debugging sessions WinDbg can start to only remote debugging sessions and loading dump files.

Restricted mode can be enabled by Windows Defender Application Control (WDAC) policy or by registry key.

Configuration using Windows Defender Application Control (WDAC)

Restricted mode can be enabled by Windows Defender Application Control (WDAC) policy. WDAC policy can prevent local administrators from altering policy settings after a policy has been deployed. To enable restricted mode by WDAC policy, configure your policy with the following setting:

XML

```
<Settings>
    <!-- Other settings -->
    <Setting Provider="Microsoft.WindbgX" Key="Settings"
ValueName="EnableRestrictedMode">
        <Value>
            <Boolean>true</Boolean>
        </Value>
    </Setting>
</Settings>
```

Configuration using Registry Key

To enable restricted mode by registry key, set this registry key

`HKLM\SOFTWARE\Microsoft\WinDbg\EnableRestrictedMode` to the DWORD value of 1.

This example command shows how to use the [reg add command](#) to add the key and set it to a value of 1.

PowerShell

```
PS C:\WINDOWS\system32> reg add HKLM\SOFTWARE\Microsoft\WinDbg /v  
EnableRestrictedMode /t REG_DWORD /d 1  
The operation completed successfully.
```

See Also

[WinDbg Features](#)

[WinDbg – Command line startup options](#)

WinDbg Basics

Article • 10/25/2023



Title	Description
WinDbg - Release notes	What's new with WinDbg

The debugger data model

Title	Description
dx command	Interactive command to display a debugger object model expression
Using LINQ With the debugger objects	SQL like query language
Native Debugger Objects in NatVis	Using the objects with NatVis
WinDbg - Data model	How to use the built in data model support in WinDbg

Extending the data model

Title	Description
JavaScript Debugger Scripting	How to use JavaScript to create scripts that understand debugger objects
WinDbg - Scripting	Using the WinDbg built in scripting
https://github.com/Microsoft/WinDbg-Samples ↗	The debugger team GitHub site where they share the latest JavaScript (and C++) sample code.
Native Debugger Objects in JavaScript Extensions	Describes how to work with common objects and provides reference information on their attributes and behaviors.

TTD Basics

Title	Description
Time Travel Debugging - Overview	TTD Overview
Time Travel Debugging - Sample App Walkthrough	To give time travel a try checkout this tutorial

TTD queries

Title	Description
Introduction to Time Travel Debugging objects.	You can use the data model to query time travel traces.
https://github.com/Microsoft/WinDbg-Samples/blob/master/TTDQueries/tutorial-instructions.md	A tutorial on how to debug C++ code using TTD queries to find the problematic code
https://github.com/Microsoft/WinDbg-Samples/tree/master/TTDQueries/app-sample	All of the code used in the lab is available here.

Videos

Watch these episodes of [Defrag Tools](#) to see WinDbg in action.

Title	Description
Defrag Tools #182	Tim, Chad, and Andy go over the basics of WinDbg and some of the features
Defrag Tools #183	Nick, Tim, and Chad use WinDbg and go over a quick demo
Defrag Tools #184	Bill and Andrew walk through the scripting features in WinDbg
Defrag Tools #185	James and Ivette provide an introduction to Time Travel Debugging
Defrag Tools #186	James and JCAB covers advanced Time Travel Debugging

Installation and getting connected

Title	Description
WinDbg – Installation	Installation directions
WinDbg – Start a user-mode session	User Mode

Title	Description
WinDbg – Start a kernel mode session	Kernel Mode

Time Travel Debugging - Overview

Article • 11/15/2024



What is Time Travel Debugging?

Time Travel Debugging is a tool that allows you to capture a trace of your process as it executes and then replay it later both forwards and backwards. Time Travel Debugging (TTD) can help you debug issues easier by letting you "rewind" your debugger session, instead of having to reproduce the issue until you find the bug.

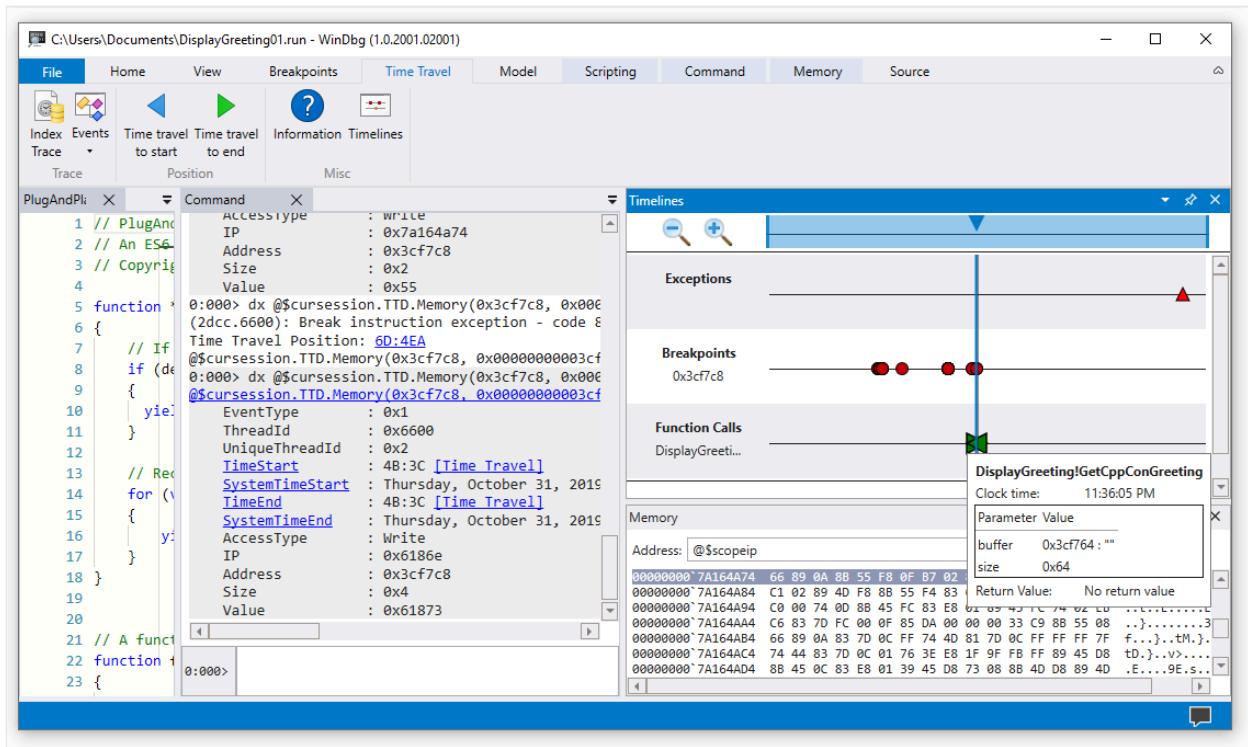
TTD allows you to go back in time to better understand the conditions that lead up to the bug and replay it multiple times to learn how best to fix the problem.

TTD can have advantages over crash dump files, which often miss the state and execution path that led to the ultimate failure.

In the event you can't figure out the issue yourself, you can share the trace with a coworker and they can look at exactly what you're looking at. This can allow for easier collaboration than live debugging, as the recorded instructions are the same, whereas the address locations and code execution will differ on different PCs. You can also share a specific point in time to help your coworker figure out where to start.

TTD is efficient and works to add as little as possible overhead as it captures code execution in trace files.

TTD includes a set of debugger data model objects to allow you to query the trace using LINQ. For example, you can use TTD objects to locate when a specific code module was loaded or locate all of the exceptions.



Requirements

Time Travel Debugging is integrated with [WinDbg](#), providing seamless recording and replay experience.

To use TTD, you need to run the debugger elevated. Install WinDbg using an account that has administrator privileges and use that account when recording in the debugger. In order to run the debugger elevated, select and hold (or right-click) the WinDbg icon in the Start menu, and then select **More > Run as Administrator**.

Release notes

TTD continues to evolve, for the latest information, see [Time travel debugging release notes](#).

The recording may contain personally identifiable or security related information

The created trace file that contains the recording may contain personally identifiable or security related information, including but not necessarily limited to file paths, registry, memory or file contents. Exact information depends on target process activity while it was recorded. Be aware of this when sharing recording files with other people.

TTD.exe command line recording utility

In addition to recording traces in the WinDbg UI, there is a TTD.exe command line utility available to record a trace.

You may have scenarios where only the TTD command line recorder is required: recording on a PC without installing the debugger, advanced recording scenarios, test automation, etc. In these scenarios you can install just the TTD command line recorder through a URL. For more information, see [Time Travel Debugging - TTD.exe command line utility](#).

Comparison of Debugging Tools

This table summarizes the pros and cons of the different debugging solutions available.

[+] Expand table

Approach	Pros	Cons
Live debugging	Interactive experience, sees flow of execution, can change target state, familiar tool in familiar setting.	Disrupts the user experience, may require effort to reproduce the issue repeatedly, may impact security, not always an option on production systems. With repro difficult to work back from point of failure to determine cause.
Dumps	No coding upfront, low-intrusiveness, based on triggers.	Successive snapshot or live dumps provide a simple "over time" view. Overhead is essentially zero if not used.
Telemetry & logs	Lightweight, often tied to business scenarios / user actions, machine learning friendly.	Issues arise in unexpected code paths (with no telemetry). Lack of data depth, statically compiled into the code.
Time Travel Debugging (TTD)	Great at complex bugs, no coding upfront, offline repeatable debugging, analysis friendly, captures everything.	Large overhead at record time. May collect more data than is needed. Data files can become large.

Video Training

To learn more about TTD see these videos.

[Defrag Tools 185](#) - Ivette and JamesP go over the basics of TTD and demo some features in WinDbg

[Defrag Tools 186](#) - Jordi and JCAB demo more great features of TTD in WinDbg

[CppCon \(YouTube\)](#) - Jordi, Ken and JamesM presented TTD in WinDbg at CppCon 2017

Trace file basics

Trace file size

The trace file can get big and the user of TTD needs to make sure that there is adequate free space available. If you record a program for even a few minutes, the trace files can quickly grow to be several gigabytes. TTD doesn't set a maximum size of trace files to allow for complex long running scenarios. Quickly re-creating the issue, will keep the trace file size as small as possible.

Trace and index files

A trace file (`.run`) stores the code execution during recording.

Once the recording is stopped, an index file (`.idx`) is created to optimize access to the trace information. Index files are also created automatically when WinDbg opens trace files.

Index files can also be large, typically twice as large as the trace file.

You can recreate the index file from the trace file using the `!tt.index` command.

```
dbgcmd
0:000> !tt.index
Successfully created the index in 10ms.
```

Recording errors and other recording output is written to a WinDbg log file.

All of the output files are stored in a location configured by the user. The default location is in the users document folder. For example, for User1 the TTD files would be stored here:

```
Console
C:\Users\User1\Documents
```

For more information on working the trace files, see [Time Travel Debugging - Working with trace files](#).

Things to look out for

Anti-virus incompatibilities

You may encounter incompatibilities because of how TTD hooks into process to record them. Typically issues arise with anti-virus or other system software that is attempting to track and shadow system memory calls. If you run into issues of with recording, such as an insufficient permission message, try temporarily disabling any anti-virus software.

Other utilities that attempt to block memory access, can also be problematic, for example, the Microsoft Enhanced Mitigation Experience Toolkit.

Another example of an environment that conflicts with TTD, would be the electron application framework. In this case the trace may record, but a deadlock or crash of the process being recorded is also possible.

User mode only

TTD currently supports only user mode operation, so tracing a kernel mode process is not possible.

Read-only playback

You can travel back in time, but you can't change history. You can use read memory commands, but you can't use commands that modify or write to memory.

System Protected Processes

Some Windows system protected processes, such as Protected Process Light (PPL) process are protected, so the TTD cannot inject itself into the protected process to allow for the recording of the code execution.

Performance impact of recording

Recording an application or process impacts the performance of the PC. The actual performance overhead varies based upon the amount and type of code being executed during recording. You can expect about a 10x-20x performance hit in typical recording

scenarios. Sometimes there will not be a noticeable slowdown in the UI. But for the more resource intensive operations, such as the File Open dialog, you will see the impact of recording.

Trace file errors

There are some cases where trace file errors can occur. For more information, see [Time Travel Debugging - Troubleshooting](#).

Advanced Features of Time Travel Debugging

Timelines

Timelines are a visual representation of events that happen during the execution. These events can be locations of: breakpoints, memory read/writes, function calls and returns, and exceptions. For more information about timelines, see [WinDbg - Timelines](#).

Debugger data model support

- **Built in data model support** - TTD includes data model support. Using LINQ queries to analyze application failures can be a powerful tool. You can use the data model window in WinDbg to work with an expandable and browsable version of 'dx' and 'dx -g', letting you create tables using NatVis, JavaScript, and LINQ queries.

For general information about the debugger data model, see [WinDbg - Data model](#). For information about working with the TTD debugger object model, see [Time Travel Debugging - Introduction to Time Travel Debugging objects](#).

Scripting support

- **Scripting Automation** - Scripting support for JavaScript and NatVis allows for the automation of problem investigation. For more information, see [Time Travel Debugging - JavaScript Automation](#).

For general information about working with JavaScript and NatVis, see [WinDbg - Scripting](#).

TTD.exe Command line utility

The TTD.exe command line utility to record traces is available. For more information, see [Time Travel Debugging - TTD.exe command line utility](#).

Managed code TTD support

You can use the SOS debugging extension (sos.dll) running in 64 bit mode to debug managed code using TTD in WinDbg. For more information, see [Debugging Managed Code Using the Windows Debugger](#).

Getting started with TTD

Review these topics to record and replay a trace file as well as for information on working with trace files and troubleshooting.

- [Time Travel Debugging - Record a trace](#)
- [Time Travel Debugging - Replay a trace](#)
- [Time Travel Debugging - Working with trace files](#)
- [Time Travel Debugging - Troubleshooting](#)
- [Time Travel Debugging - Sample App Walkthrough](#)

These topics describe additional advanced functionality in time travel debugging.

- [Time Travel Debugging - Introduction to Time Travel Debugging objects](#)
- [Time Travel Debugging - JavaScript Automation](#)

Feedback

Was this page helpful?



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

Time travel debugging release notes

Article • 11/15/2024



This topic provides information on what's new in Time Travel Debugging.

1.11.429

This update of TTD contains a few bug fixes along with some internal changes to improve reliability.

Note: 1.11.410 introduced a regression in the emulation of the Intel/AMD LODSD instruction. A fix for this will come in the next release.

Fixes:

- Improve packet reading robustness and other misc changes to improve reliability.
- Fix a regression in emulating the AVX VBROADCAST[I/F]128 instruction.
- Fix the exception record access on ARM64 in newer builds of Windows.

1.11.410

Improved accessibility: Progress UI now properly scales with Text Size changes.

The `@$cursession.TTD.Calls()` command in the debugger now supports wildcards that match a large number of functions. It is now possible to query for large numbers of functions (`@$cursession.TTD.Calls("kernel32!*")`).

Automation: A new `-onMonitorReadyEvent` command-line option indicates when the recording monitor (`-monitor` switch) is ready to record new processes.

Fixes:

- Fix some race conditions while initializing the recorder.
- Fix how we record syscalls so that breakpoints work correctly.
- Fix multiple issues related to module selective recording.

ARM64 fixes:

- Fixed a bug preventing TTD recording on plain ARM64v8.0 level CPUs.
- Improved the messaging when attempting to use on ARM64 a trace of an x86 or x64 process.

AMD/Intel fixes (includes some issues reported by Google):

- Fixed incorrect emulation of LODS: Instead of zeroing out the unused bits of RAX they are now correctly preserved.
- Fixed emulation of "pop ax" instruction in x86/x64 processes, which was incorrectly zeroing the upper bits of the full register (e.g. "pop ax" cleared the upper bits of rax).
- Direct emulation of the XGETBV instruction (faster).
- Direct emulation of all AVX512 SIMD moves (faster).

1.11.316

Fixed a regression that was causing occasional crashes when recording programs with long uninterrupted data-heavy instruction sequences.

ARM64 fixes:

- Recording in ARM64 processes that have the PAC feature enabled is now supported.
- Fixed the ANDS and TST instructions, which were failing to clear the carry and overflow flags.

AMD/Intel fixes:

- Fixed bug where TTD incorrectly emulated "xchg r8,rax" and "xchg r8w,ax" as NOP.

1.11.304

TTD now implements and publishes publicly an API to control the recorder from within the live recorded process. Documentation and a sample can be found in [GitHub ↗](#).

TTD can now inject itself with recording turned off using the new `-recordMode` switch. By default TTD uses `-recordMode Automatic` which causes all threads to be recorded. If `-recordMode Manual` is specified then TTD injects into the target process but doesn't record anything until told to do so through an API call.

Recording can now be restricted to a specific set of modules using the `-module` switch. In some scenarios this can result in substantially faster recording and smaller trace files.

More than one `-module` switch may be specified.

Matching record and replay components are now included in the distribution. In the event of an incompatibility between the debugger and the command line recorder, or a replay bug, the replay components can be copied into the debugger install as a workaround until a new debugger is released.

The installed file location can be found in Powershell by doing the following:

```
ls (Get-AppxPackage | where Name -eq  
'Microsoft.TimeTravelDebugging').InstallLocation
```

Added

- Add `-recordmode` switch to enable injection without automatic recording (1.11.296)
- Add `-module` switch and use to create SR config (1.11.291)
- Project custom data recorded by the in-process API to data model (1.11.286)
- Add a new `TTDLiveRecorder.dll` and wire it up along with `TTDRecordCPU.dll` (1.11.283)
- Add replay components to MSIX & fix SDK lookup (1.11.265)

Changed

none

Fixed

- Work around [a bug ↴](#) in the nlohmann JSON serializer, which is used in some internal tooling (1.11.281)
 - Contributed [a fix ↴](#) to that library that will become available in [a future release ↴](#).
- Adjust string alignment to avoid a rare CRT bug (1.11.279)
 - Reported to and fixed in the VS and OS codebases.
- Several small fixes from Watson crash reports (1.11.276)
- Fix a regression that may cause trace file corruption in some cases (1.11.264)

Known issues

- On ARM64, the compiler is failing to tail-call a number of high-frequency functions which in extreme cases can cause the recorder to run out of stack space and crash.

1.11.261

Notable changes in this release include:

- [ARM64] Fixed the behavior of the `SXTL`, `SQXTN2`, `SQXTUN2`, `UQXTN2`, `XTN2` and `TRN1` instructions when the destination register is used as a source.
- [ARM64] Fixed an issue that caused the debugger to show SIMD registers with their lower 64 bits duplicated into the high 64 bits.
- [AMD64] AVX512 emulation fixes for AMD's Zen4 processors (registers were getting corrupted).

Changed

- Implement a new versioning system specifically for the emulator. (1.11.260)

Fixed

- Fix faulty ARM64 instructions where the destination register is also used as a source. (1.11.261)
- Fix the Zen4 workaround for direct returns to the emulator. (1.11.222)

1.11.202

This release fixes a number of issues encountered while recording services or monitoring process launch via the `-monitor` switch. It also removes ARM32 recording support from the product.

Changed

- Reduce binary size by linking TTD components to UCRT as a DLL. (1.11.191)

Fixed

- Fix recording of services. (1.11.193)
- Fix several issues when using `-monitor`. (1.11.189)
- Fix the stack frame of the function that preserves non-volatiles when running fallbacks on x64 (1.11.188)

- Re-enabled output buffering for TTD (1.11.187)
- Fix the handling of the GPO's handle in the ProcessMonitorServer (1.11.179)

Removed

- Remove ARM32 recording code from the repo (1.11.198)

1.11.173

This release increases visibility of certain error messages by extracting them from the .out file and printing them to the console. It also fixes a rare crash during trace replay.

Changed

- Extract and print error messages from .out file (1.11.173)

Fixed

- Fix file conflict while reading .out file from separate process. (1.11.171)
- Fix rare crash during trace replay. (1.11.166)

1.11.163

This release adds support for recording x86 processes on x64 machines.

Changed

- Fix x86 recording with x64 TTD installation (1.11.163)

Fixed

- EULA cleanup (1.11.161)

1.11.159

This release is the first public release of the command line recorder. Along with several changes required to enable public release of the command line recorder, this release also includes a number of bug fixes, including a couple of fixes to the CPU emulator.

The new `-timestampFileName` switch enables timestamp-based .run file generation. This is useful when you are recording many instances of the same process, and want to minimize recording startup time.

Changed

- Choose default injection mode at runtime based on which tracer is used (1.11.156)
- Add switch to enable timestamp-based .run file generation (1.11.155)
- Add EULA and `-accepteula` to TTD (1.11.154)
- Add ProcLaunchMon.sys to MSIX (1.11.153)
- Create per-arch MSIX and MSIXBUNDLE (1.11.152)
- Fix a number of issues that came up when testing TTD built with Clang. (1.11.146)
- Clang fixes for TTDAAnalyze (1.11.144)

Fixed

- Review feedback on appinstaller / public release (1.11.159)
- RC feedback (1.11.157)
- Avoid trashing the Zero register by initializing RegisterInfo to point to Sink. (1.11.149)
- Fix TST instruction with immediate and enhance the unit test to cover it and more. (1.11.148)
- Consolidate protected process decision and disable protected process use (1.11.147)

1.11.138

Changed

- Create recorder MSIX (1.11.138)
- Fix all issues so Clang can build TTD. (1.11.137)
- Introduce `-monitor X` as way to record a process when it launches (1.11.116)

Fixed

- Fix "CMP ZR" ARM64 emulation (1.11.128)
- Fix AVX512 emulation on AMD's Zen4 processors (1.11.127)
- Fix the mechanism TTD uses to find files for a specific CPU (1.11.121)
- Fix x86 TTD regression (TTDRecordCPU.dll fails to load) (1.11.110)

- Fix the return to native path on ARM64 to not trash X28 (1.11.109)

See Also

[Time Travel Debugging – Overview](#)

[Time Travel Debugging – Command line recorder](#)

Feedback

Was this page helpful?



Yes



No

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

Time Travel Debugging - Record a trace

Article • 11/18/2024

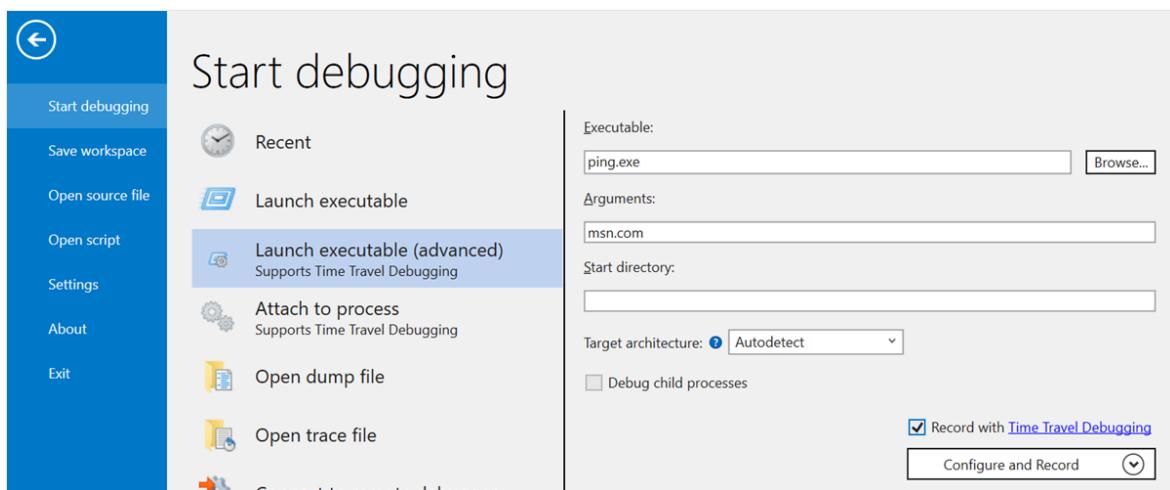


This section describes how to record time travel debugging (TTD) traces. There are two ways to record a Trace in WinDbg, *Launch Executable (advanced)* and *Attach to a process*.

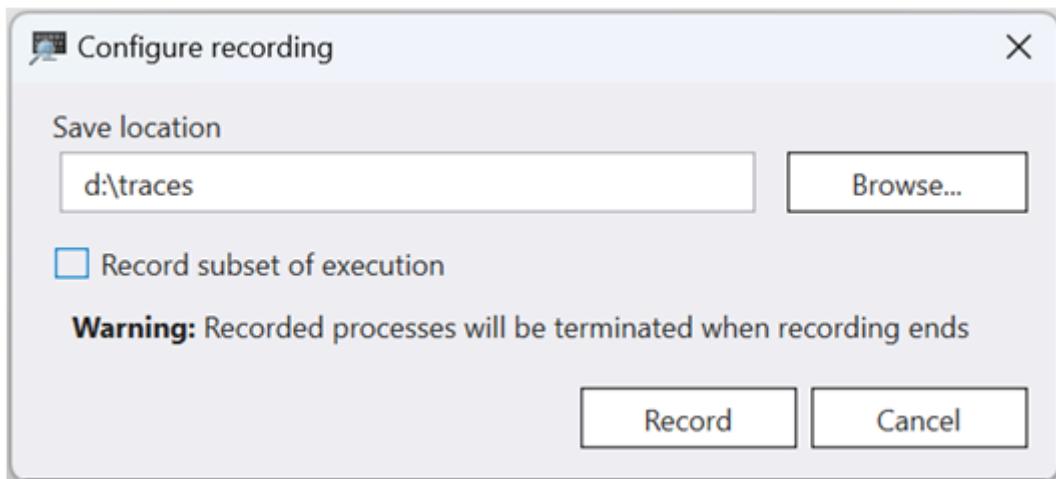
Launch executable (advanced)

To launch an executable and record a TTD trace, follow these steps.

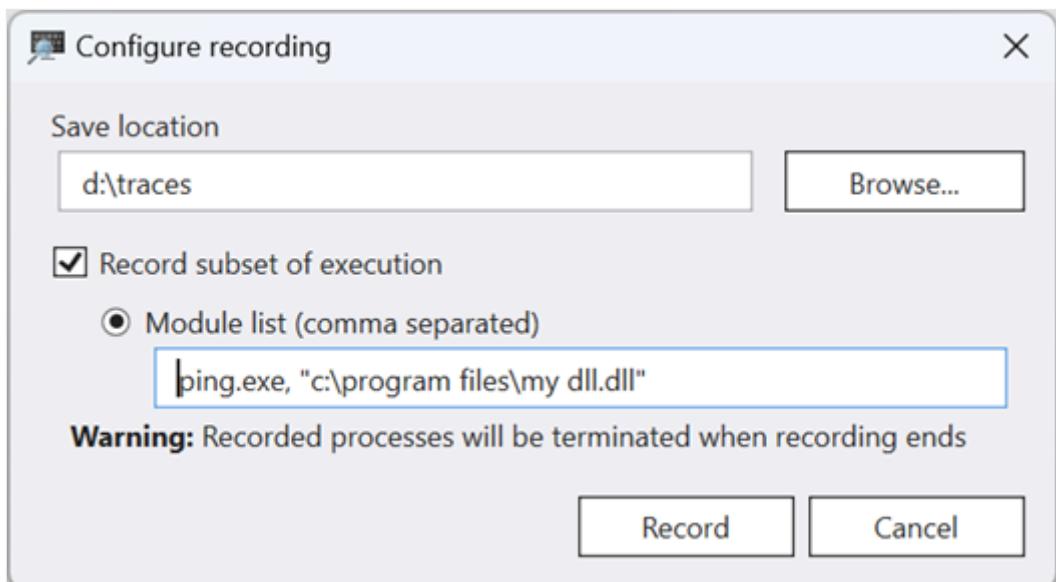
1. In WinDbg, select **File > Start debugging > Launch executable (advanced)**.
2. Enter the path to the user mode executable that you wish to record or select **Browse** to navigate to the executable. For information about working with the Launch Executable menu in WinDbg, see [WinDbg - Start a user-mode session](#).
3. Check the **Record with Time Travel Debugging** box to record a trace when the executable is launched.



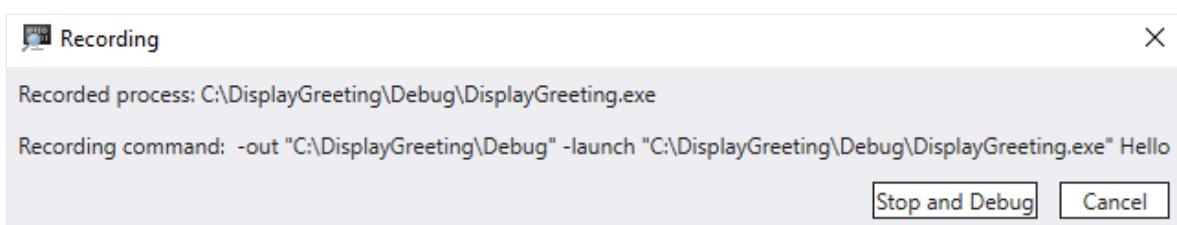
4. If you select **Configure and Record** you will be able to configure a location for the trace file.



5. To limit recording to specific modules, check "Record subset of execution" and type in the module names. For example, if you only want to record the execution of notepad.exe, type "notepad.exe" in the text box. If you want to record the execution of notepad.exe and kernelbase.dll, type "notepad.exe,kernelbase.dll" in the text box.



6. Select **OK** to launch the executable and start recording.
7. The recording dialog appears indicating the trace is being recorded.

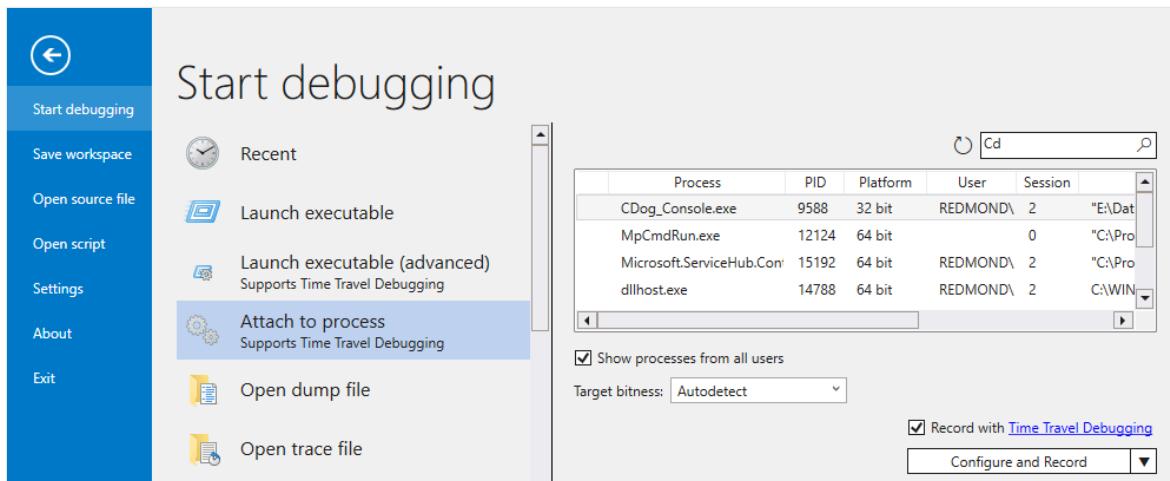


8. See [How to record](#) for information on recording.

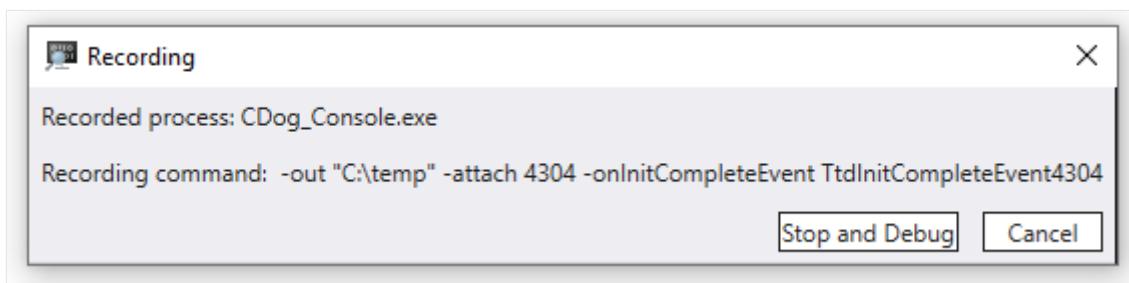
Attach to a process

To attach to a process and record a TTD trace, follow these steps.

1. In WinDbg, select **File > Start debugging > Attach to process**.
2. Select the user mode process that you wish to trace. For information about working with *Attach to a process* menu in WinDbg, see [WinDbg - Start a user-mode session](#).



3. Check the **Record Process with Time Travel Debugging** box to create a trace when the executable is launched.
4. Select **Attach** to start recording.
5. The recording dialog appears indicating the trace is being recorded.



6. See [How to record](#) for information on recording.

How to record

1. The process is being recorded, so this is where you need to cause the issue that you wish to debug. You may open a problematic file or select a specific button in the app to cause the event of interest to occur.
2. While the recording dialog box is being displayed you can:
 - **Stop and debug** - Choosing this will stop the recording, create the trace file and open the trace file so you can start debugging.

- **Cancel** - Choosing this will stop the recording and create the trace file. You can open the trace file at a later time.

3. Once your recording is complete, close your app or hit **Stop and debug**.

 **Note**

Both *Stop and debug* and *Cancel* will terminate the associated process.

4. When the application being recorded terminates, the trace file will be closed and written out to disk. This is the case if your program crashes as well.

5. When a trace file is opened, the debugger will automatically index the trace file. Indexing allows for more accurate and faster memory value look ups. This indexing process will take longer for larger trace files.

```
dbgcmd  
...  
00007ffc`61f789d4 c3          ret  
0:000> !index  
Indexed 1/1 keyframes  
Successfully created the index in 96ms.
```

 **Note**

A keyframe is a location in a trace used for indexing. Keyframes are generated automatically. Larger traces will contain more keyframes. When the trace is indexed, the number of keyframes is displayed.

6. At this point you are at the beginning of the trace file and are ready to travel forward and backward in time.

 **Tip**

Using breakpoints is a common approach to pause code execution at some event of interest. Unique to TTD, you can set a breakpoint and travel back in time until that breakpoint is hit after the trace has been recorded. The ability to examine the process state after an issue has happened, to determine the best location for a breakpoint, enables additional debugging workflows. For

an example of using a breakpoint in the past, see [Time Travel Debugging - Sample App Walkthrough](#).

Next Steps

Now that you have recorded a TTD trace, you can replay the trace back or work with the trace file, for example sharing it with a co-worker. For more information, see these topics.

[Time Travel Debugging - Replay a trace](#)

[Time Travel Debugging - Working with trace files](#)

[Time Travel Debugging - Troubleshooting](#)

[Time Travel Debugging - Sample App Walkthrough](#)

See Also

[Time Travel Debugging - Overview](#)

Feedback

Was this page helpful?



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

Time Travel Debugging - Replay a trace

Article • 11/18/2024



This section describes how to replay time travel traces, navigating forwards and backwards in time.

Command time travel navigation

Use a trailing minus sign with the following commands to travel back in time.

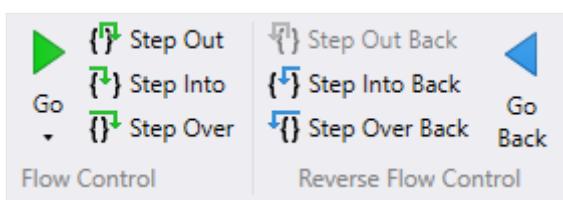
[] Expand table

Command
p- (Step Back)
t- (Trace Back)
g- (Go Back)

For more information, see [Time Travel Debugging - Navigation commands](#).

Ribbon button time travel navigation

Alternatively, use the ribbon buttons to navigate in the trace.



Example TTD Trace Replay

Use the g- command to execute backwards until either an event or the beginning of the TTD trace is reached. The events that can stop backward execution are the same that would stop forward execution. In this example, the start of the trace is reached.

```
dbgcmd
```

```
0:000> g-
TTD: Start of trace reached.
(3f78.4274): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 29:0
ntdll!ZwTestAlert+0x14:
00007ffc`61f789d4 c3          ret
```

Use the [p \(Step\)](#) command to step forward in a TTD trace.

```
dbgcmd
```

```
0:000> p
Time Travel Position: F:1
ntdll!LdrpInitializeProcess+0x1bc5:
7774f828 740b      je      ntdll!LdrpInitializeProcess+0x1bd2
(7774f835) [br=1]
0:000> p
Time Travel Position: F:2
ntdll!LdrpInitializeProcess+0x1bd2:
7774f835 83bdd0feffff00  cmp     dword ptr [ebp-130h],0
ss:002b:010ff454=00000000
0:000> p
Time Travel Position: F:3
ntdll!LdrpInitializeProcess+0x1bd9:
7774f83c 0f8450e8ffff  je      ntdll!LdrpInitializeProcess+0x42f
(7774e092) [br=1]
```

You can also use the [t \(Trace\)](#) command to navigate in the trace.

```
dbgcmd
```

```
0:000> t
Time Travel Position: F:4
ntdll!LdrpInitializeProcess+0x42f:
7774e092 33c0      xor     eax,eax
0:000> t
Time Travel Position: F:5
ntdll!LdrpInitializeProcess+0x431:
7774e094 e9f5170000  jmp     ntdll!LdrpInitializeProcess+0x1c2b
(7774f88e)
```

Use the p- command to step backwards in a TTD trace.

```
dbgcmd
```

```
0:000> p-
Time Travel Position: F:4
```

```
ntdll!LdrpInitializeProcess+0x42f:  
7774e092 33c0          xor      eax,eax  
0:000> p-  
Time Travel Position: F:3  
ntdll!LdrpInitializeProcess+0x1bd9:  
7774f83c 0f8450e8ffff je      ntdll!LdrpInitializeProcess+0x42f  
(7774e092) [br=1]
```

You can also use the t- command to navigate backwards in time.

!tt navigation commands

Use the !tt command to navigate forward or backwards in time, by skipping to a given position in the trace.

!tt [position]

Provide a time position in any of the following formats to travel to that point in time.

- If [position] is a decimal number between 0 and 100, it travels to approximately that percent into the trace. For example `!tt 50` travels to halfway through the trace.
- If {position} is #:#, where # are a hexadecimal numbers, it travels to that position. For example, `!tt 1A0:12F` travels to position 1A0:12F in the trace.

For more information, see [Time Travel Debugging - !tt \(time travel\)](#).

!positions

Use `!positions` to display all the active threads, including their position in the trace. For more information, see [Time Travel Debugging - !positions \(time travel\)](#).

```
dbgcmd  
  
0:000> !positions  
>*Thread ID=0x1C74 - Position: F:2  
Thread ID=0x1750 - Position: A5:0  
Thread ID=0x3FFC - Position: 200:0  
Thread ID=0x36B8 - Position: 403:0  
Thread ID=0x3BC4 - Position: 5F2:0  
Thread ID=0x392C - Position: B45:0  
Thread ID=0x32B4 - Position: C87:0  
Thread ID=0x337C - Position: DF1:0  
* indicates an actively running thread
```

This example shows that there are eight threads at the current position. The current thread is 3604, marked with '>'.

Tip

Another way to display the current list of threads and their positions, is to use the data model dx command:

```
dx -g @$curprocess.Threads.Select(t => new { IsCurrent = t.Id ==  
@$curthread.Id, ThreadId = t.Id, Position = t.TTD.Position })
```

Use the user mode [~ \(Thread Status\)](#) command shows the same eight threads, and marks the current thread with ':':

```
dbgcmd
```

```
0:000> ~  
. 0  Id: 954.1c74 Suspend: 4096 Teb: 00fdb000 Unfrozen  
    1  Id: 954.1750 Suspend: 4096 Teb: 00fea000 Unfrozen  
    2  Id: 954.3ffc Suspend: 4096 Teb: 00fde000 Unfrozen  
    3  Id: 954.36b8 Suspend: 4096 Teb: 00fe1000 Unfrozen  
    4  Id: 954.3bc4 Suspend: 4096 Teb: 00fe4000 Unfrozen  
    5  Id: 954.392c Suspend: 4096 Teb: 00fed000 Unfrozen  
    6  Id: 954.32b4 Suspend: 4096 Teb: 00ff0000 Unfrozen  
    7  Id: 954.337c Suspend: 4096 Teb: 00ff3000 Unfrozen
```

In the !positions command output, click on the link next to the third thread (3FFC), to time travel to that position in the trace, 200:0.

```
dbgcmd
```

```
0:002> !tt 200:0  
Setting position: 200:0  
(954.3ffc): Break instruction exception - code 80000003 (first/second chance  
not available)  
Time Travel Position: 200:0  
ntdll!NtWaitForWorkViaWorkerFactory+0xc:  
7775396c c21400          ret      14h
```

Use the [~ \(Thread Status\)](#) command to confirm that we are now positioned at the third thread, 3ffc.

```
dbgcmd
```

```
0:002> ~  
0  Id: 954.1c74 Suspend: 4096 Teb: 00fdb000 Unfrozen
```

```
1 Id: 954.1750 Suspend: 4096 Teb: 00fea000 Unfrozen
. 2 Id: 954.3ffc Suspend: 4096 Teb: 00fde000 Unfrozen
3 Id: 954.36b8 Suspend: 4096 Teb: 00fe1000 Unfrozen
4 Id: 954.3bc4 Suspend: 4096 Teb: 00fe4000 Unfrozen
5 Id: 954.392c Suspend: 4096 Teb: 00fed000 Unfrozen
6 Id: 954.32b4 Suspend: 4096 Teb: 00ff0000 Unfrozen
7 Id: 954.337c Suspend: 4096 Teb: 00ff3000 Unfrozen
```

ⓘ Note

The `~s#`, where `#` is a thread number, also switches to the given thread, but it doesn't change the current position in the trace. When `!tt` is used to time travel to another thread's position, any values you (and the debugger) read from memory will be looked up at that position. When switching threads with `~s#`, the debugger doesn't change the current position internally, which is used for all memory queries. This works this way primarily so that `~s#` doesn't have to reset the debugger's inner loop.

Time travel debugging extension commands

For information on the `!tt`, `!positions` and the `!index` commands see [Time Travel Debugging - Extension Commands](#).

See Also

[Time Travel Debugging - Overview](#)

[Time Travel Debugging - Record a trace](#)

[Time Travel Debugging - Working with trace files](#)

[Time Travel Debugging - Sample App Walkthrough](#)

Feedback

Was this page helpful?

 Yes

 No

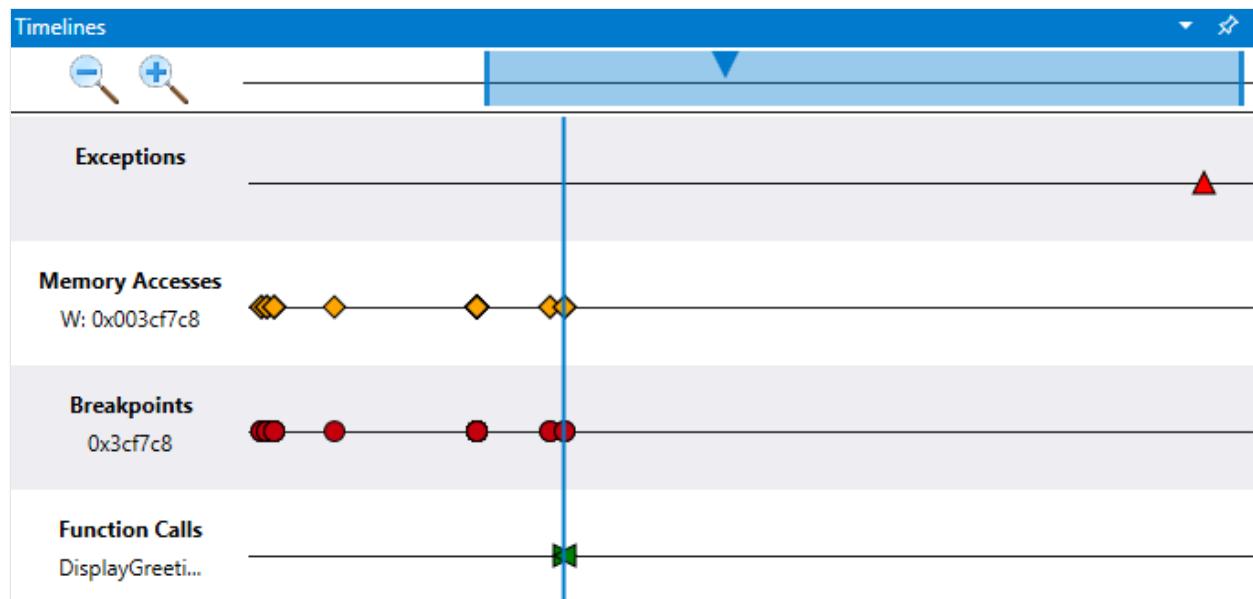
[Provide product feedback ↗](#) | Get help at Microsoft Q&A

WinDbg - Timelines

Article • 03/14/2024



Time Travel Debugging (TTD) allows users to record traces, which are recordings of the execution of a program. Timelines are a visual representation of events that happen during the execution. These events can be locations of: breakpoints, memory read/writes, function calls and returns, and exceptions.



Use the timelines window to quickly view important events, understand relative position and easily jump to their location in your TTD trace file. Use multiple timelines to visually explore events in the time travel trace and discover event correlation.

The timelines window is displayed when opening a TTD trace file and shows key events without you having to manually create data model queries. At the same time, all of the time travel objects are available to allow for more complex data queries.

For more information about creating and working with Time Travel trace files, see [Time Travel Debugging - Overview](#).

Types of Timelines

The timelines window can display the following events:

- Exceptions (you can further filter on a specific exception code)
- Breakpoints

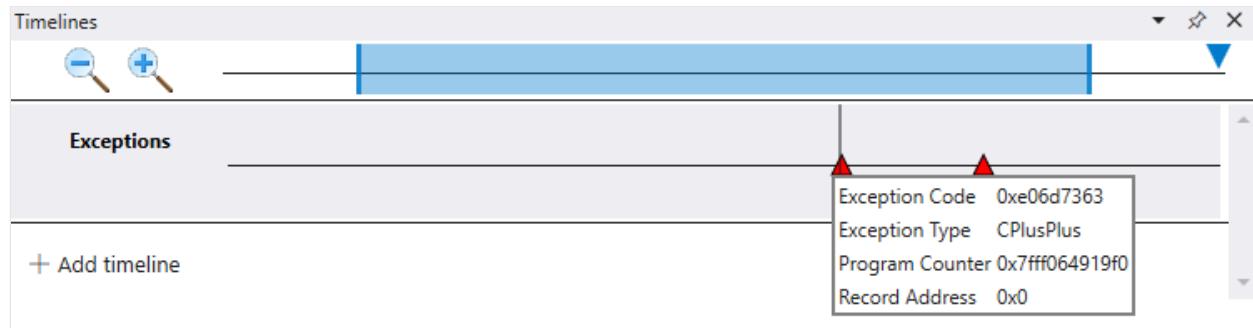
- Function Calls (search in the form of module!function)
- Memory Accesses (read / write / execute between two memory addresses)

Hover over each event to get more information via tooltip. Clicking on an event will run the query for the event and display more information. Double-clicking an event will jump to that location in the TTD trace file.

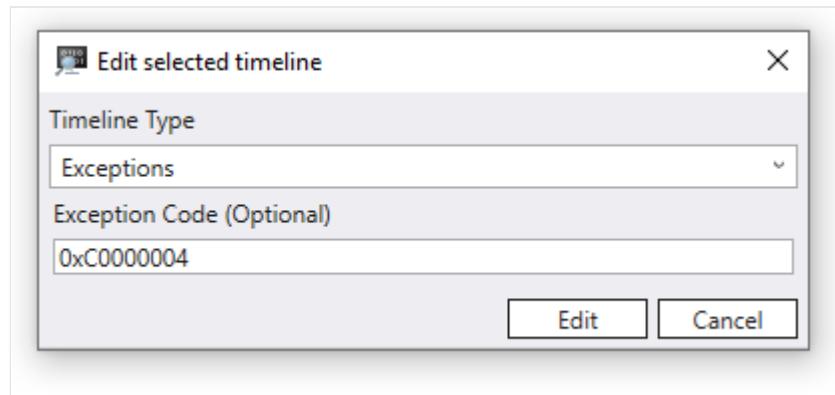
Exceptions

When you load a trace file and the timeline is active, it will display any exceptions in the recording automatically.

When you hover over a breakpoint information such as the exception type and the exception code are displayed.



You can further filter on a specific exception code using the optional exception code field.



You can also add a new timeline for a specific exception type.

Breakpoints

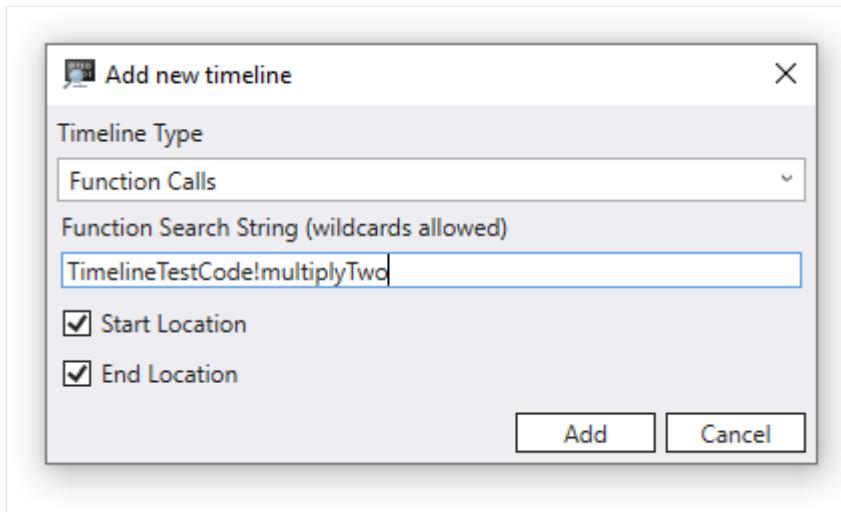
After adding a breakpoint, you can display the positions of when that breakpoint is hit on a timeline. This can be done for example using the [bp Set Breakpoint command](#). When you hover over a breakpoint the address and the instruction pointer associated with the breakpoint is displayed.



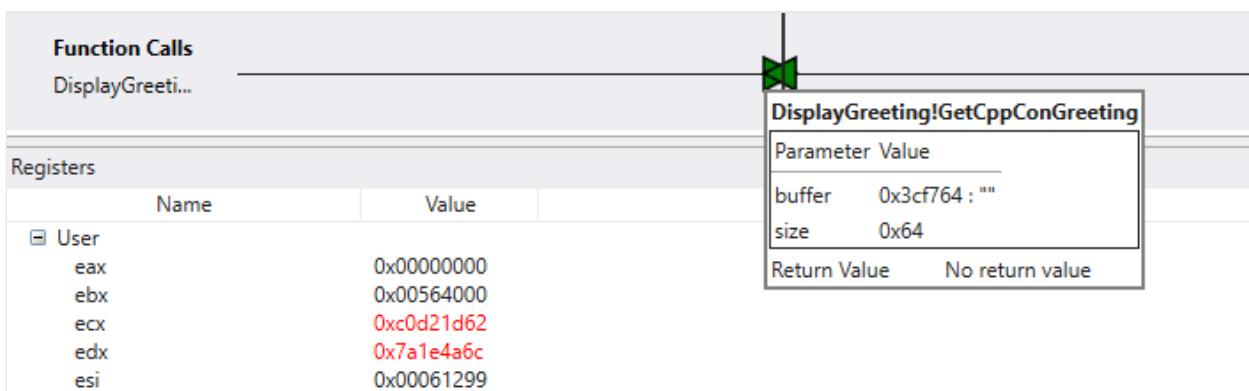
When the breakpoint is cleared, the associated breakpoint timeline is automatically removed.

Function Calls

You can display the positions of function calls on the timeline. To do this provide the search in the form of `module!function`, for example `TimelineTestCode!multiplyTwo`. You can also specify wildcards, for example `TimelineTestCode!*m*`.

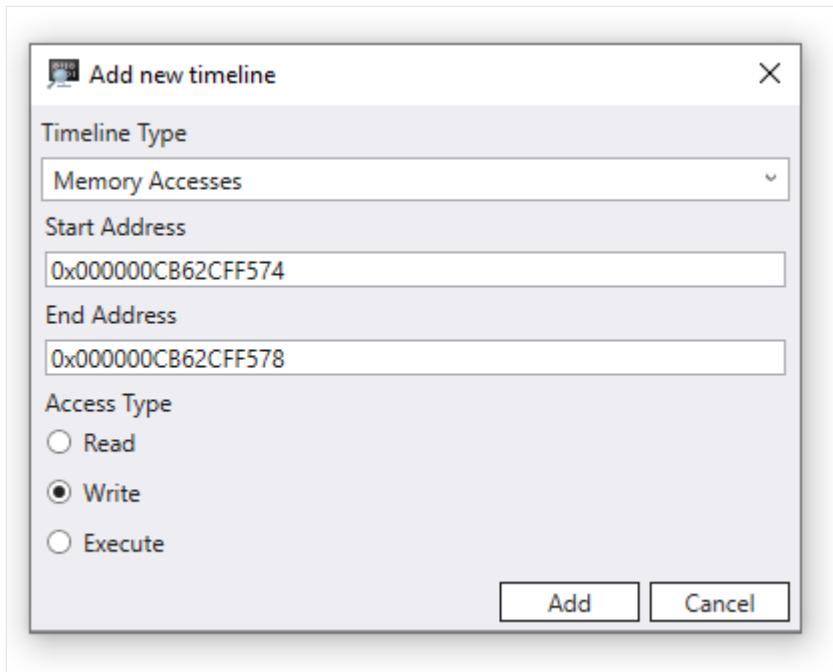


When you hover over a function call the function name, input parameters, their values, and the return value are displayed. This example shows *buffer* and *size* since those are the parameters to `DisplayGreeting!GetCppConGreeting`.

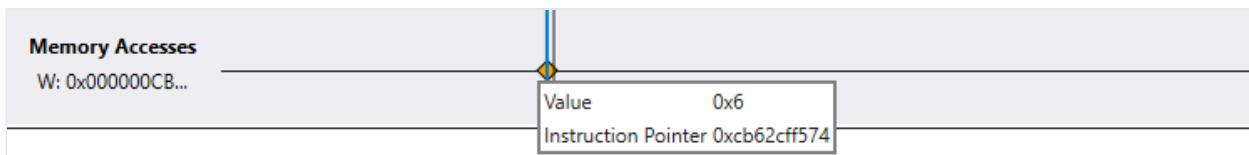


Memory Access

Use the memory access timeline to display when a specific range of memory has been read or written to, or where code execution has taken place. A start and stop address is used to define a range between two memory addresses.



When you hover over a memory access item the value and the instruction pointer are displayed.

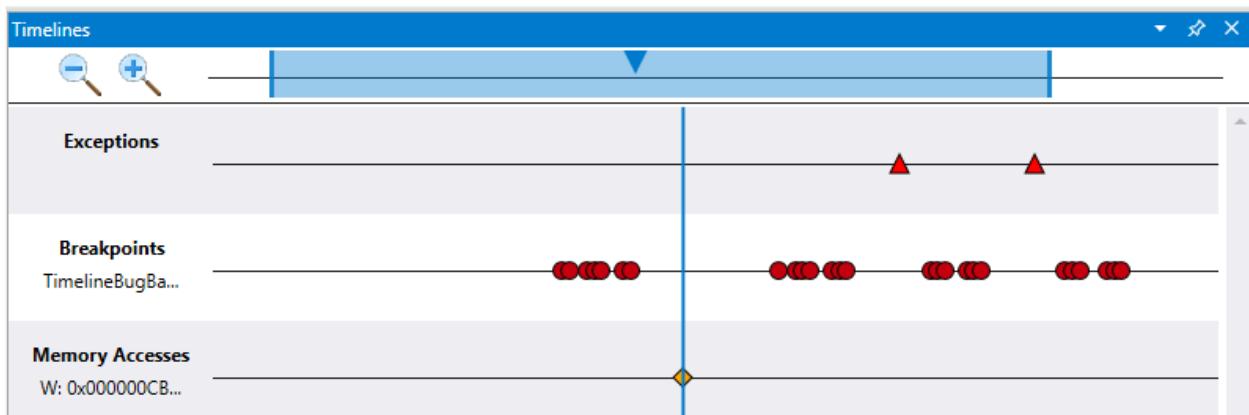


Work with timelines

A vertical gray line follows the cursor when hovering over the timeline. The vertical blue line indicates the current position in the trace.

Click on the magnifying glass icons to zoom in and out on the timeline.

In the top timeline control area use the rectangle to pan the view of the timeline. Drag the outer delimiters of the rectangle to resize the current timeline view.



Mouse Movements

Zoom in and out using Ctrl + Scroll wheel.

Pan from side to side using Shift + Scroll wheel.

Timeline debugging techniques

To demonstrate debugging timeline techniques, the [Time Travel Debugging Walkthrough](#) is reused here. This demonstration assumes that you have completed the first two steps to build the sample code and created the TTD recording using the first two steps described there.

[Section 1: Build the sample code](#)

[Section 2: Record a trace of the "DisplayGreeting" sample](#)

In this scenario, the first step is to find the exception in the time travel trace. This can be done by double clicking on the only exception that is on the timeline.

Looking in the command window we see the following command was issued when we clicked on the exception.

```
dbgcmd

(2dcc.6600): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: CC:0
@$curprocess.TTD.Events.Where(t => t.Type == "Exception")
[0x0].Position.SeekTo()
```

Select **View >> Registers** to display the registers at this point in the timeline to begin our investigation.

The screenshot shows the Visual Studio debugger interface with three main windows:

- Timelines**: Shows a timeline with a blue bar representing the recording. There are search and zoom icons on the left.
- Exceptions**: Shows a list of exceptions. A red arrow points to the first entry in the list.
- Registers**: A table showing register values:

Name	Value
esp	0x003cf7d0
ebp	0x00520055
eip	0x00540020
efl	0x000000246

In the command output note that the stack (esp) and base pointer (ebp) are pointing to two very different addresses. This could indicate that stack corruption - possibly a function returned and then corrupted the stack. To validate this, we need to travel back to before the CPU state was corrupted and see if we can determine when the stack corruption occurred.

As we do that, we will examine the values of local variables and the stack.

Select **View >> Locals** to display the local values.

Select **View >> Stack** to display the code execution stack.

At the point of failure in trace it is common to end up a few steps after the true cause in error handling code. With time travel we can go back an instruction at a time, to locate the true root cause.

From the **Home** ribbon use the **Step Into Back** command to step back three instructions. As you do this, continue to examine the stack, locals and register windows.

The command window will display the time travel position and the registers as you step back three instructions.

```
dbgcmd

0:000> t-
Time Travel Position: CB:41
eax=00000000 ebx=00564000 ecx=c0d21d62 edx=7a1e4a6c esi=00061299
edi=00061299
eip=00540020 esp=003cf7d0 ebp=00520055 iopl=0          nv up ei pl zr na pe
nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
00540020 ??          ???
0:000> t-
Time Travel Position: CB:40
eax=00000000 ebx=00564000 ecx=c0d21d62 edx=7a1e4a6c esi=00061299
edi=00061299
eip=00061767 esp=003cf7cc ebp=00520055 iopl=0          nv up ei pl zr na pe
nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
DisplayGreeting!main+0x57:
00061767 c3          ret
0:000> t-
Time Travel Position: CB:3A
eax=0000004c ebx=00564000 ecx=c0d21d62 edx=7a1e4a6c esi=00061299
edi=00061299
eip=0006175f esp=003cf718 ebp=003cf7c8 iopl=0          nv up ei pl nz na pe
nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
```

```

efl=00000206
DisplayGreeting!main+0x4f:
0006175f 33c0          xor     eax,eax

```

At this point in the trace our stack and base pointer have values that make more sense, so it appears that we have getting closer to the point in the code where the corruption occurred.

```

dbgcmd

esp=003cf718 ebp=003cf7c8

```

Also of interest is that the locals window contains values from our target app and the source code window is highlighting the line of code that is ready to be executed in our source code at this point in the trace.

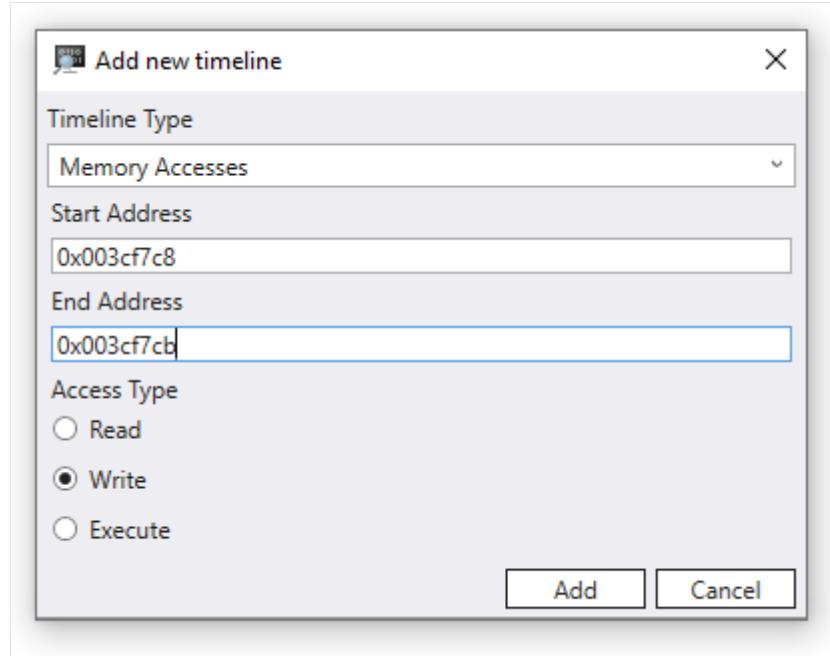
To further investigate, we can open up a memory window to view the contents near the stack pointer (esp) memory address. In this example it has a value of 003cf7c8. Select **Memory >> Text >> ASCII** to display the ASCII text stored at that address.

The screenshot shows the Immunity Debugger interface with three main windows:

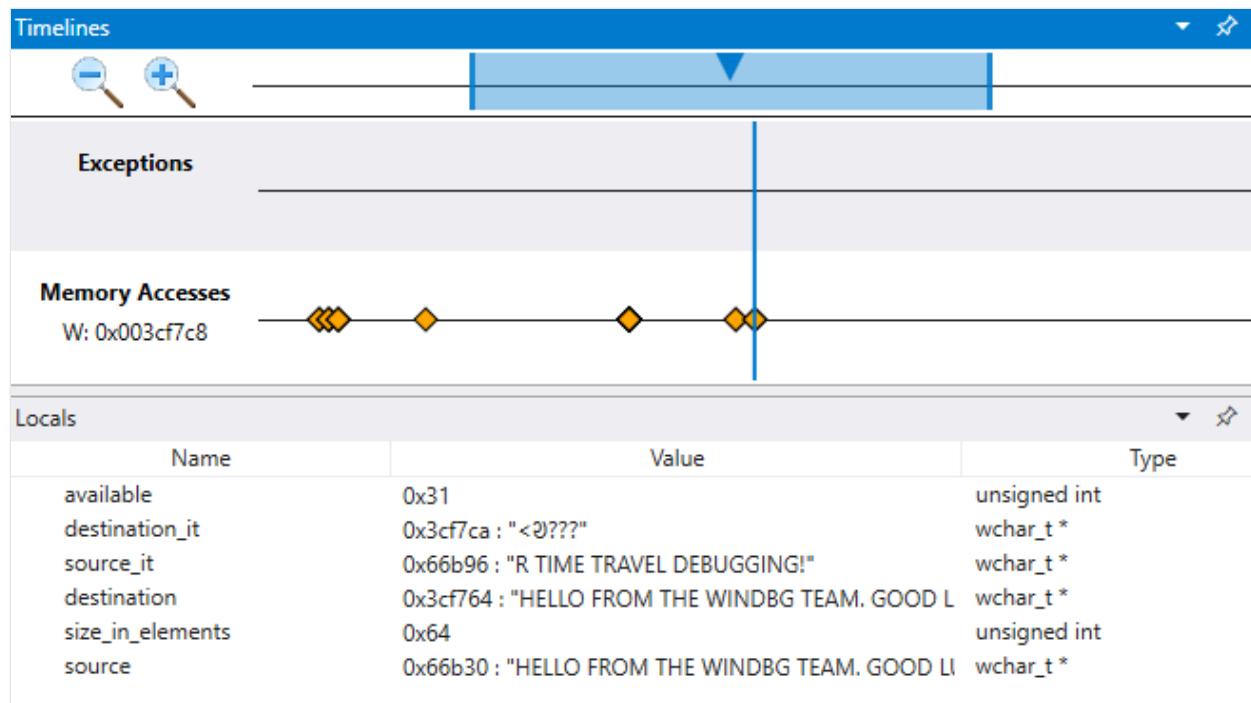
- Registers** window: Shows the current register values. esp is highlighted in red, indicating it is the current stack pointer.
- Locals** window: Shows the local variables and their values. The 'greeting' variable is expanded, showing its size and the ASCII values of its four elements: 'H', 'E', 'L', and 'L'.
- Memory** window: Shows the memory dump starting at address 0x003cf7c8. The address is selected in the dropdown. The dump area shows the memory layout, with some bytes highlighted in red, corresponding to the values in the Locals window.

Memory access timeline

After a memory location of interest has been identified, add a memory access timeline using that value. Click on **+ Add timeline** and fill in the starting address. We will look at 4 bytes, so adding that to the start address of 003cf7c8, we have 003cf7cb. The default is to look at all memory writes, but you can also look at just writes or code execution at that address.



We can now traverse the timeline in reverse to examine at what point in this time travel trace this memory location was written to see what we can find. Clicking on this position in the timeline we see that locals value different values for the string being copied. The destination value appears to not be complete, as if the length of our string is not correct.



Breakpoint timeline

Using breakpoints is a common approach to pause code execution at some event of interest. TTD allows you to set a breakpoint and travel back in time until that breakpoint is hit after the trace has been recorded. The ability to examine the process state after an issue has happened, to determine the best location for a breakpoint, enables additional debugging workflows unique to TTD.

To explore an alternative timeline debugging technique, click on the exception in the timeline and once again travel three steps back, using the **Step Into Back** command on the **Home** ribbon.

In this very small sample it would be pretty easy to just look in the code, but if there are hundreds of lines of code and dozens of subroutines the techniques described here can be used to decrease the time necessary to locate the issue.

As mentioned earlier, the base pointer (esp) instead of pointing to an instruction, is pointing to our message text.

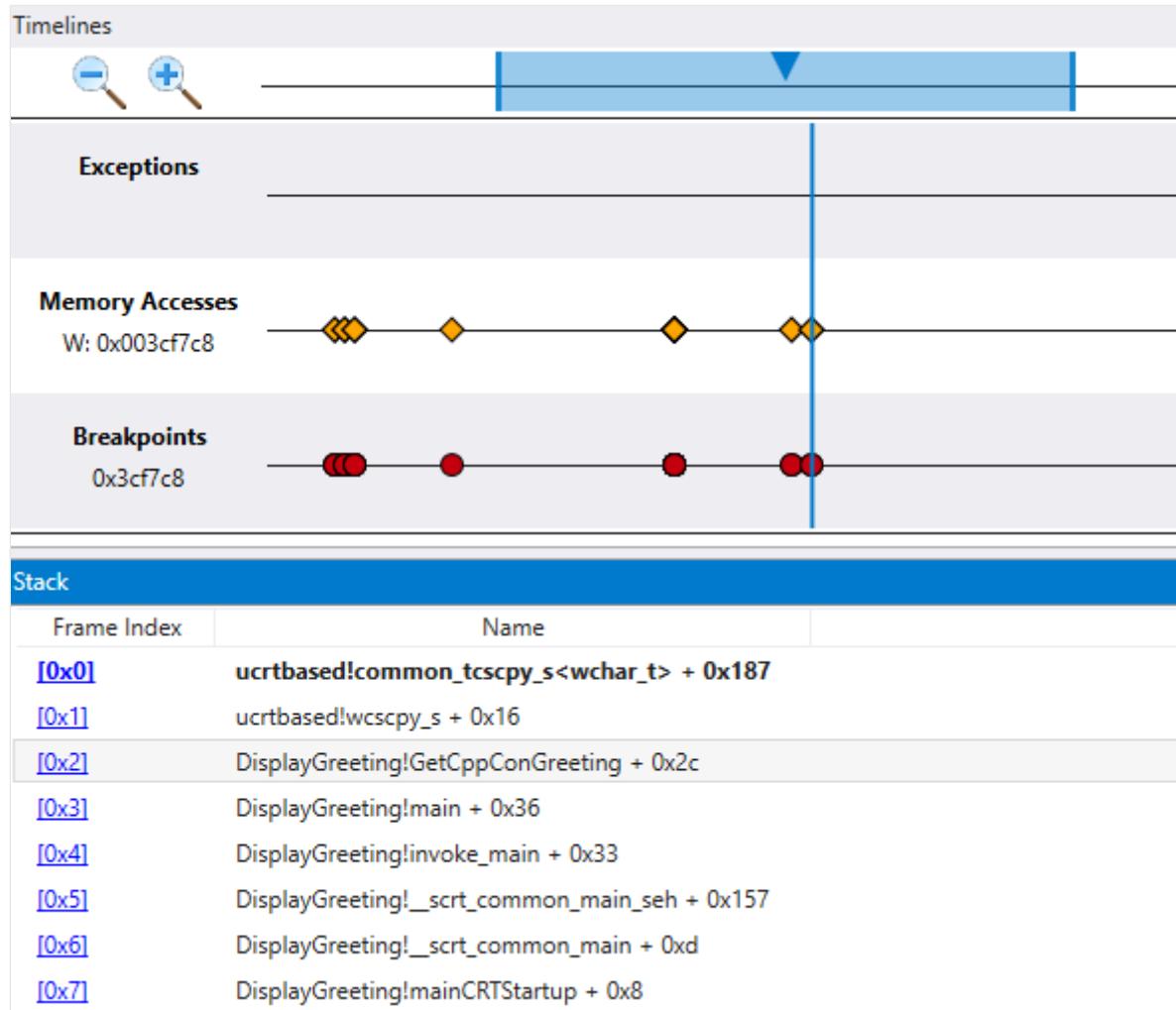
Use the `ba` command to set a breakpoint on memory access. We will set a `w` - write breakpoint to see when this area of memory is written to.

```
dbgcmd
0:000> ba w4 003cf7c8
```

Although we will use a simple memory access breakpoint, breakpoints can be constructed to be more complex conditional statements. For more information, see [bp](#), [bu](#), [bm](#) ([Set Breakpoint](#)).

From the Home menu, select **Go Back** to travel back in time until the breakpoint is hit.

At this point we can examine the program stack to see what code is active.



As it is very unlikely that the Microsoft provided wscopy_s() function would have a code bug like this, we look further in the stack. The stack shows that Greeting!main calls Greeting!GetCppConGreeting. In our very small code sample we could just open the code at this point and likely find the error pretty easily. But to illustrate the techniques that can be used with larger, more complex program, we will set add a function call timeline.

Function call timeline

Click on **+ Add timeline** and fill in the `DisplayGreeting!GetCppConGreeting` for the function search string.

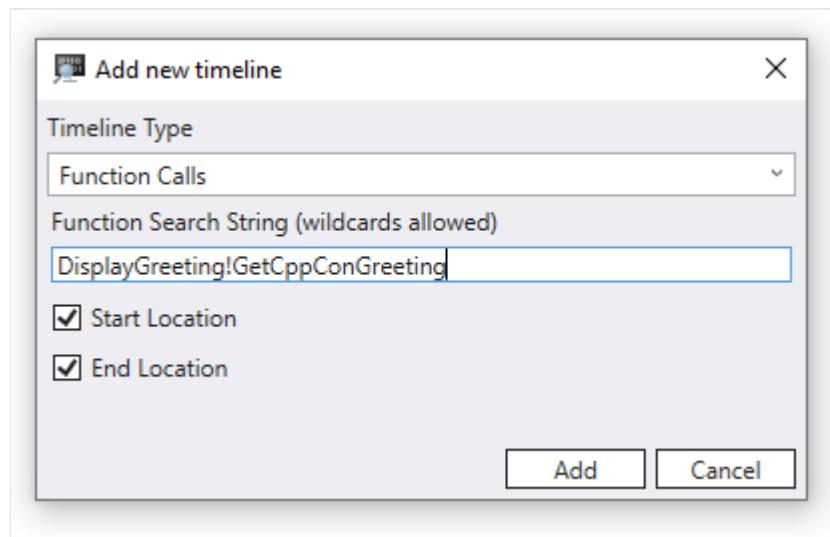
The Start and End location check boxes indicate that the start and end of a function call in the trace.

We can use the dx Command to display the function call object to see the associated TimeStart and TimeEnd fields which corresponds to Start Location and End Location of the function call.

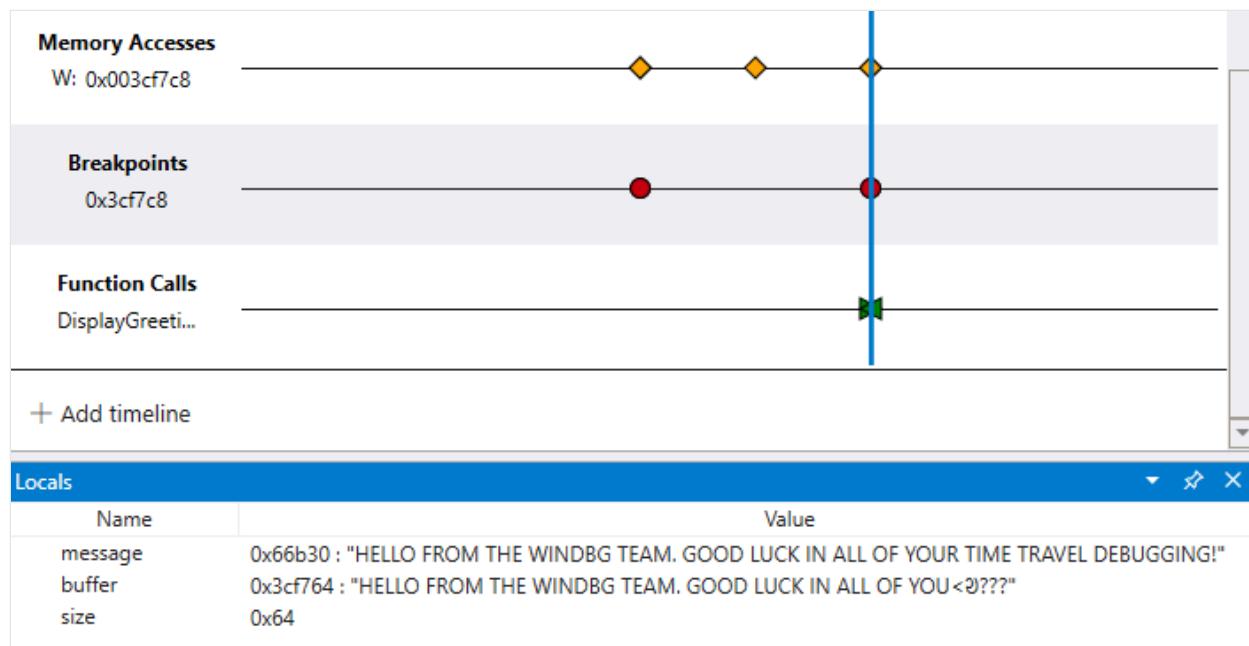
```
dbgcmd

dx @$cursession.TTD.Calls("DisplayGreeting!GetCppConGreeting")[0x0]
  EventType      : 0x0
  ThreadId       : 0x6600
  UniqueThreadId : 0x2
  TimeStart       : 6D:BD [Time Travel]
  SystemTimeStart : Thursday, October 31, 2019 23:36:05
  TimeEnd         : 6D:742 [Time Travel]
  SystemTimeEnd   : Thursday, October 31, 2019 23:36:05
  Function        : DisplayGreeting!GetCppConGreeting
  FunctionAddress  : 0x615a0
  ReturnAddress    : 0x61746
  Parameters
```

Either the Start or End, or both the Start and End location boxes, must be checked.

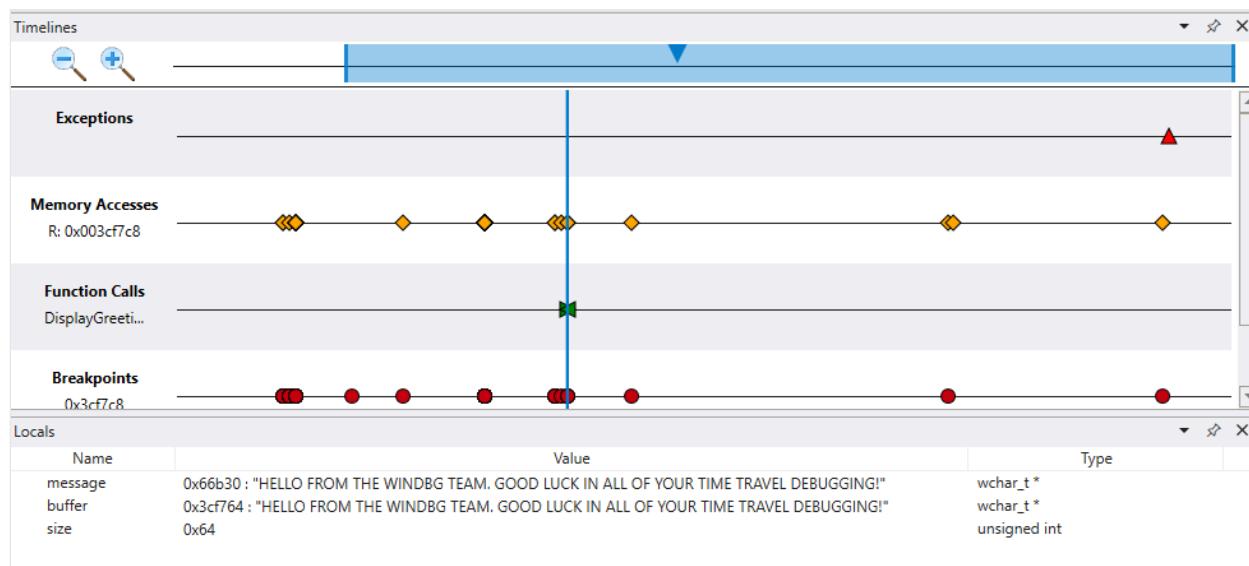


As our code is neither recursive or re-entrant, it is pretty easy to locate on the time line when the GetCppConGreeting method is called. The call to GetCppConGreeting also occurs at the same time as our breakpoint as well as the memory access event that we defined. So it looks like we have narrowed in on an area of code to look carefully at for the root cause of our application crash.



Explore code execution by viewing multiple timelines

Although our code sample is small, the technique of using multiple timelines allows for visual exploration of a time travel trace. You can look across the trace file to ask questions, such as "when is an area of memory accessed before a breakpoint is hit?".

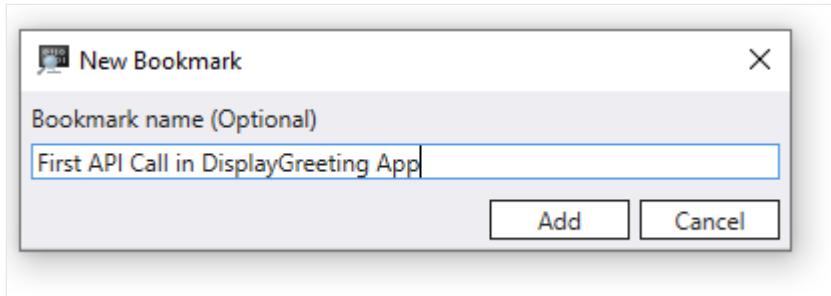


The ability to see additional correlations and find things you may not have expected, differentiates the timeline tool from interacting with the time travel trace using command line commands.

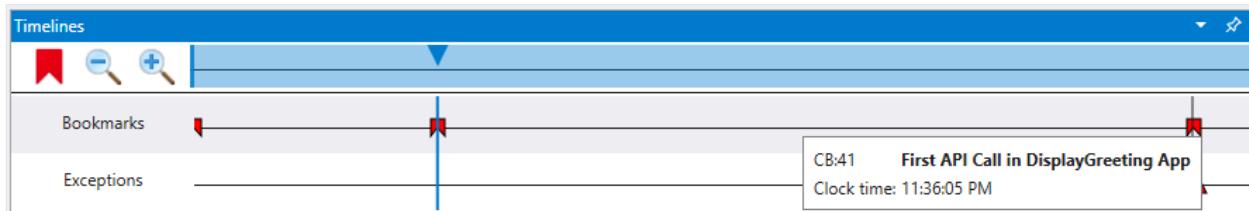
Timeline Bookmarks

Bookmark important Time Travel positions in WinDbg instead of manually copy pasting the position to notepad. Bookmarks make it easier to view at a glance different positions in the trace relative to other events, and to annotate them.

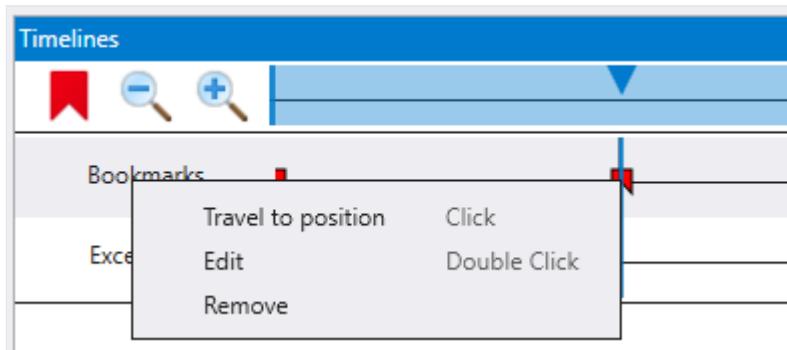
You can provide a descriptive name for bookmarks.



Access Bookmarks via the Timeline window available in *View > Timeline*. When you hover over a bookmark, it will display the bookmark name.



You can right click the bookmark to travel to that position, rename or delete the bookmark.



ⓘ Note

In version 1.2402.24001.0 of the debugger the bookmark feature is not available.

See Also

[WinDbg Features](#)

[Time Travel Debugging Walkthrough](#)

Time Travel Debugging - Working with Trace Files

Article • 11/15/2024



This section describes how to work with files created and consumed by time travel debugging.

Trace File Overview

Time Travel Debugging uses the following files to debug code execution.

- The trace file contains the code execution recording and has a .RUN extension.
- The index file enables quick access to information in the Trace file and has an .IDX extension.
- Recording errors and other recording output is written to the debugger log file.

Trace .RUN files

Trace .RUN files can be opened after they are recorded using **File > Start debugging > Open trace file**.

Start debugging



Recent



Launch executable



Launch executable (advanced)

Supports Time Travel Debugging



Attach to process

Supports Time Travel Debugging



Open dump file



Open trace file

All of the trace output files are stored in the users document folder by default. For example, for User1 the TTD files would be stored here:

Console

C:\Users\User1\Documents

You can change the location of the trace files when you start to record. For more information, see [Time Travel Debugging - Recording](#).

The most recently used list of files allows you to quickly access previously used target configuration files. Any recently used trace files or dump files are listed as well.

Today



C:\MyTTDTraces\Ping01.run

Open dump file

9/12/2017 3:31 PM ↗



C:\MyTTDTraces\Write01.run

Open dump file

9/12/2017 3:30 PM ↗



C:\MyTTDTraces\CDog_Console01.run

Open dump file

9/12/2017 3:14 PM ↗



C:\MyTTDTraces\Notepad01.run

Open dump file

9/12/2017 3:11 PM ↗

Index .IDX files

An index .IDX file is created for the associated trace .RUN file automatically when opening the trace file in WinDbg. You can manually create the index file by using the !index command. An index allows for faster access to the trace information.

IDX files can also be large, typically twice the size of the .RUN file.

Recreating the .IDX file

You can recreate the .IDX file from the .RUN file, using the `!index` command. For more information, see [Time Travel Debugging - !index \(time travel\)](#).

```
dbgcmd  
0:0:001> !index  
Indexed 3/3 keyframes  
Successfully created the index in 49ms.
```

Sharing TTD Trace .RUN files

TTD is local only and doesn't work remotely connected to another machine.

TTD trace files can be shared with others by copying the .RUN file. This can be handy for having a coworker help you figure out the problem. They don't need to install the crashing app or do any other related setup to attempt to reproduce the issue. They can just load the trace file and debug the app as if it was installed on their PC.

The machine where you replay the TTD trace must support all instructions that were used on the record machine - for example AVX instructions.

You can rename the file to include any additional information, such as the date or a bug number.

The .IDX file doesn't need to be copied as it can be re-created using the `!index` command as described above.

💡 Tip

When collaborating with others, pass on any relevant trace positions related to the problem at hand. The collaborator can use the `!tt x:y` command to move to that

exact point in time in the execution of the code. Time position ranges can be included in bug descriptions to track where the possible issue may be occurring.

Error - Log File

Recording errors and other recording output is written to the debugger log file. To view the log file, select **View > Logs**.

This example shows the error log text when attempting to launch and record an executable named Foo.exe that is not in the C:\Windows directory.

```
Console

2017-09-21:17:18:10:320 : Information : DbgXUI.dll : TTD: Output:
Microsoft (R) TTD 1.01.02
Release: 10.0.16366.1000
Copyright (C) Microsoft Corporation. All rights reserved.
Launching C:\Windows\Foo.exe
2017-09-21:17:18:10:320 : Error : DbgXUI.dll : TTD: Errors:
Error: Trace of C:\Windows\Foo.exe PID:0 did not complete successfully:
status:27
Error: Could not open 'Foo.exe'; file not found.
Error: Corrupted trace dumped to C:\Users\User1\Documents\Foo01.run.err.
```

Trace file size

TTD trace files can get really big and it is important to ensure that you have adequate free disk space available. If you record an app or process even just for a few minutes, the trace file may grow to be several gigabytes in size. The size of the trace file is dependent on a number of factors described below.

TTD doesn't set a maximum size of trace files to allow for complex long running scenarios. Quickly re-creating the issue, will keep the trace file size as small as possible.

Trace file size factors

It is not possible to provide an exact estimate of the trace file size but there are a few rules of thumb to help you understand TTD file sizes.

The following factors can affect the size of the trace file:

- The number of code instructions executed across all threads when recording the running app or process

- The length of time during which the app or process was recorded (only as this affects the number of code instructions recorded)
- The size of the memory data used by the app or process

The number of instructions executed and recorded is the biggest factor that affects the trace file size. A trace typically requires between 1 bit and 1 byte per instruction executed. A trace is likely to be towards the lower end of that range when the recorded program executes a smaller number of distinct functions and operates on a smaller set of data. A trace is likely to be towards the higher end of that range when the recorded program executes a larger number of distinct functions or operates on a larger set of data.

Trace file size rule of thumb

The trace file grows roughly 5 MB to 50 MB per second when recording an active app or process, depending on the trace file size factors identified above.

The trace file will not grow when the app or process being recorded is idle (for example when waiting for input).

Currently there is no maximum file size limit for trace files. WinDbg can replay trace files that are well into the hundreds of gigabytes in size.

Index file size

The index file is automatically created by WinDbg when you open a trace for the first time. It contains information that helps the debugger to replay the trace and query information about memory more efficiently. Its size generally ranges from 1 to 2 times the size of the trace file. The factors that affect its size are similar to those that affect the size of the trace file.

First, the size of the index file scales relative to the length of the trace. A trace that contains a larger number of recorded instructions will generally have a larger index.

Second, the size of the index scales relative to the breadth of the memory accesses. If the program that was recorded frequently accessed a large number of distinct memory locations, the index will generally be larger than if the program that was recorded accessed fewer distinct memory locations or if accesses to memory locations was less frequent.

Since these factors are similar to the factors that affect the size of the trace file, the size of the index file generally scales relative to the size of the trace file (thus our estimate that it's typically between 1x and 2x the size of the trace file).

What if I run out of disk space?

Both the TTD trace and index files are written to disk. Currently there is not a maximum file size limitation for either the trace or index file. The trace file grows in size until you stop recording or exceed the amount of available disk space.

During recording: TTD will write out last page to the trace file and then effectively wait until it can write again. WinDbg continues to show the Recording dialog but doesn't show an error/warning message when running out of disk space during recording.

Running out of disk space during recording results in a trace file with an incomplete record of the code execution. The incomplete trace file can be opened in WinDbg but it may not include the actual problem if the error occurs after running out of disk space when writing the trace file.

Workaround: Open File Explorer and check if the disk free space is near zero. Alternately watch the trace (.RUN) file in File Explorer (default in Documents folder) and if not regularly growing in size then recording may be waiting. Select the Stop and Debug button in WinDbg, free up space or save to another disk, and start recording again.

During indexing: The debugger may produce an invalid index file, resulting in unpredictable behavior in the debugger, or the debugger engine host may crash.

Workaround: Close the debugger and delete any index file (.idx) that may exist for your trace. Either free up sufficient disk space, or move the trace file to a different disk with sufficient free space. Open the trace again in the debugger and run !index to create a new, correct index. Indexing doesn't modify the original trace file (.run), so no data will have been lost.

See Also

[Time Travel Debugging - Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time Travel Debugging - Troubleshooting

Article • 11/15/2024



This section describes how to troubleshoot time travel traces.

Issues attempting to record a process

I get an error message that says "WinDbg must be run elevated to support Time Travel Debugging"

As the message indicates, running the debugger elevated is a requirement. In order to run the debugger elevated, right-click on the **WinDbg** icon in the start menu and then select **More > Run as Administrator**.

I installed WinDbg with an account that doesn't have administrator privileges and I get an error message that says "WinDbg must be run elevated to support Time Travel Debugging"

Re-install WinDbg using an account that has administrator privileges and use that account when recording in the debugger.

I can't launch and record a UWP application

This is not supported at this time, but you may attach to and record an already-running UWP application.

I can't record a unusual process type - running in another session, security context, credentials...

At this time, TTD only record regular processes that you can be normally launched from a command console or by clicking on an executable or shortcut in Windows Explorer.

I cannot successfully record my application on my computer

If recording of your application fails, verify that you can record a simple Windows process. For example, "ping.exe" or "cmd.exe" are simple processes that can normally be recorded.

I cannot successfully record anything at all on my computer

TTD recording is an invasive technology, which can interfere with other invasive technologies like application virtualization frameworks, information management products, security software or antivirus products.

See "Things to look out for" in [Time Travel Debugging - Overview](#) for information on known TTD incompatibilities.

I'm tracing an application and running AppVerifier at the same time, and the performance when replaying the trace is slow.

Because of the way AppVerifier uses memory to check the application, the experience later when replaying the trace can be noticeably worse than without AppVerifier. To improve performance, disable AppVerifier when recording the app. If this is not possible, you may need to close the callstack window in WinDbg in order to improve performance.

Issues with .IDX index files

Debugging a trace file without an index file, or with a corrupted or incomplete index file, is possible, but is not recommended. The index file is needed to ensure that memory values read from the debugged process are most accurate, and to increase the efficiency of all other debugging operations.

Use the `!index -status` command to examine the state of the .IDX index file associated with the .RUN trace file.

If it you may try recreating the index file by running `!index -force`.

Recreating the .IDX index file

If you suspect an issue with the index file, or `!index -status` says anything other than "Index file loaded", recreate it. To do this you may run `!index -force`. If that fails:

1. Close the debugger.
2. Delete the existing IDX file, it will have the same name as the .RUN trace file and be located in the same directory that the .RUN file is.
3. Open the trace .RUN file in WinDbg. This will run the `!index` command to re-create the index.
4. Use the `!index -status` command to confirm that the trace index is functional.

Ensure that there's enough space for the index file in the same location where the trace file resides. Depending on the contents of the recording, the index file may be significantly larger than the trace file, typically on the order of twice as large.

Issues with Trace .RUN Files

When there are issues with the trace .RUN file, you may receive error messages such as these.

dbgcmd

```
Replay and log are out of sync at fallback data. Packet type is incorrect  
"Packet Type"  
Replay and log are out of sync at opaque data. Log had already reached the  
end  
Replay exit thread event does not match up with logged event  
Logged debug write values are out of sync with replay
```

In most cases all of the failure messages indicate that the .RUN trace file is not usable and must be re-recorded.

Re-recording the user mode app

If there is a specific issue with recording a user mode app, you may want to try recording a different app on the same PC, or try the same app on a different PC. You may want to try and record a different use of the app to see if there is a specific issue with recording certain parts of the app.

When debugging or creating the index, I see messages about "Derailment events".

It is possible that you may see messages like this one:

dbgcmd

```
Derailment event MissingDataDerailment(7) on UTID 2, position 2A550B:108
with PC 0x7FFE5EEB4448 Request address: 0x600020, size: 32
```

TTD works by running an emulator inside of the debugger, which executes the instructions of the debugged process in order to replicate the state of that process at every position in the recording. Derailments happen when this emulator observes some sort of discrepancy between the resulting state and information found in the trace file. The error quoted above, for instance, refers to an instruction found on location 0x7FFE5EEB4448, at position 2A550B:108 in the trace, which attempted to read some memory around location 0x600020, which doesn't exist in the recording.

Derailments are often caused by some error in the recorder, or sometimes in the emulator, at some recorded instruction further back in the trace.

In most cases this failure message indicates that the .RUN trace file will have a gap in the thread that derailed, starting at the point that it derailed, for some indeterminate number of instructions. If the event of interest you are trying to debug didn't happen during that gap, the trace may be usable. If the event of interest occurred during that gap, the trace will need to be re-recorded.

See Also

[Time Travel Debugging - Overview](#)

[Time travel debugging release notes](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time travel navigation commands

Article • 11/15/2024



This section describes the time travel navigation commands.

p- (Step Back)

The *p*- command executes the previous single instruction or source line. When subroutine calls or interrupts occur, they are treated as a single step. You can invoke this command using the **Step Over Back** button on the **Home** ribbon in WinDbg.

t- (Trace Back)

The *t*- command executes the previous single instruction or source line. When subroutine calls or interrupts occur, each of their steps is also traced. You can invoke this command using the **Step Into Back** button on the **Home** ribbon in WinDbg.

g- (Go Back)

The *g*- command starts executing the current process in reverse. Execution will halt at the end of the program, when BreakAddress is hit, or when another event causes the debugger to stop. You can invoke this command using the **Go Back** button on the **Home** ribbon in WinDbg.

Additional Information

The time travel navigation commands only work with time travel traces. For more information about time travel, see [Time Travel Debugging - Overview](#).

See Also

[Time Travel Debugging - Overview](#)

[Time Travel Debugging - Replay a trace](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time travel debugging extension commands

Article • 11/21/2024



This section introduces the time travel debugger extension commands.

In this section

[] [Expand table](#)

Topic	Description
<code>!tt (time travel)</code>	The !tt (time travel) debugger extension that allows you to navigate forward and backwards in time.
<code>!positions</code>	The !positions extension displays all the active threads, including their current positions in the time travel trace.
<code>!index</code>	The !index extension indexes time travel traces or displays index status information.

Additional Information

These extensions only works with time travel traces. For more information about time travel, see [Time Travel Debugging - Overview](#).

DLL

The time travel debugger extension commands are implemented in ttdext.dll. The time travel command dll is loaded automatically in WinDbg. You don't need to use the load command to manually load the ttdext.dll.

See Also

[Time Travel Debugging - Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

!tt (time travel)

Article • 10/25/2023



The !tt (time travel) debugger extension that allows you to navigate forward and backwards in time.

!tt navigation commands

Use the !tt extension to navigate forward or backwards in time, by traveling to a given position in the trace.

```
dbgcmd
```

```
!tt [position]
```

Parameters

position

Provide a time position in any of the following formats to travel to that point in time.

- If {position} is a decimal number between 0 and 100, it travels to approximately that percent into the trace. For example:
 - !tt 0 - Time travel to the beginning of the trace
 - !tt 50 - Time travel to halfway through the trace
 - !tt 100 - Time travel to the end of the trace
- If {position} is #:#, where # are a hexadecimal numbers, it travels to that position. If the number after : is omitted, it defaults to zero.
 - !tt 1A0: - Time travel to position 1A0:0
 - !tt 1A0:0 - Time travel to position 1A0:0
 - !tt 1A0:12F - Time travel to position 1A0:12F

ⓘ Note

Traces use a two-part instruction position that references a specific position reference in the trace, for example 12:0. or 15:7. The two elements are hexadecimal numbers defined as described here.

xx:yy

xx - the first element is the sequencing number, which corresponds to a sequencing event.

yy - the second element is a step count, which corresponds roughly to the instruction count since the sequencing event.

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see [Time Travel Debugging - Overview](#).

See Also

[Time Travel Debugging - Overview](#)

!index

Article • 11/13/2024



The `!index` extension indexes time travel traces or displays index status information.

dbgsyntax

```
!index [-status] [-force]
```

Use `!index` to run an indexing pass over the current trace.

dbgcmd

```
0:000> !index
Indexed 10/14 keyframes
Indexed 14/14 keyframes
Successfully created the index in 535ms.
```

If the current trace is already indexed, the `!index` command does nothing.

dbgcmd

```
0:000> !index
Successfully created the index in 0ms.
```

Parameters

-status

Use `!index -status` to report the status of the trace index.

dbgcmd

```
0:000> !index -status
Index file loaded.
```

-force

Use `!index -force` to reindex the trace even if an unloadable index file exists on disk.

```
dbgcmd
```

```
0:000> !index -force
Successfully created the index in 152ms.
```

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see [Time Travel Debugging - Overview](#).

See Also

[Time Travel Debugging - Extension Commands](#)

Feedback

Was this page helpful?

 Yes

 No

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

!positions

Article • 11/13/2024



The **!positions** extension displays all the active threads, including their current positions in the time travel trace.

```
dbgcmd
!positions
```

Parameters

None

Example

This output shows five threads. Thread 1660 is the current thread indicated by **>** in the left column.

```
dbgcmd
0:000> !positions
>*Thread ID=0x1660 - Position: 724:0
    Thread ID=0x3E6C - Position: A8B:0
    Thread ID=0x30EC - Position: A8A:0
    Thread ID=0x1F40 - Position: A8E:0
    Thread ID=0x4170 - Position: C44:0
* indicates an actively running thread
```

DLL

ttdext.dll

Additional Information

This extension only works with time travel traces. For more information about time travel, see [Time Travel Debugging - Overview](#).

See Also

[Time Travel Debugging - Extension Commands](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time Travel Debugging - Sample App Walkthrough

Article • 11/18/2024



This lab introduces Time Travel Debugging (TTD), using a small sample program with a code flaw. TTD is used to debug, identify and root cause the issue. Although the issue in this small program is easy to find, the general procedure can be used on more complex code. This general procedure can be summarized as follows.

1. Capture a time travel trace of the failed program.
2. Use the [dx \(Display Debugger Object Model Expression\)](#) command to find the exception event stored in the recording.
3. Use the [!tt \(time travel\)](#) command to travel to the position of the exception event in the trace.
4. From that point in the trace single step backwards until the faulting code in question comes into scope.
5. With the faulting code in scope, look at the local values and develop a hypothesis of a variable that may contain an incorrect value.
6. Determine the memory address of the variable with the incorrect value.
7. Set a memory access (ba) breakpoint on the address of the suspect variable using the [ba \(Break on Access\)](#) command.
8. Use g- to run back to last point of memory access of the suspect variable.
9. See if that location, or a few instructions before, is the point of the code flaw. If so, you are done. If the incorrect value came from some other variable, set another break on access breakpoint on the second variable.
10. Use g- to run back to the last point of memory access on the second suspect variable. See if that location or a few instructions before contains the code flaw. If so, you are done.
11. Repeat this process walking back until the code that set the incorrect value that caused the error is located.

Although the general techniques described in this procedure apply to a broad set of code issues, there are unique code issues that will require a unique approach. The techniques illustrated in the walkthrough should serve to expand your debugging tool set and will illustrate some of what is possible with a TTD trace.

Lab objectives

After completing this lab, you will be able to use the general procedure with a time travel trace to locate issues in code.

Lab setup

You will need the following hardware to be able to complete the lab.

- A laptop or desktop computer (host) running Windows 10 or Windows 11

You will need the following software to be able to complete the lab.

- The WinDbg. For information on installing WinDbg, see [WinDbg - Installation](#)
- Visual Studio to build the sample C++ code.

The lab has the following three sections.

- [Section 1: Build the sample code](#)
- [Section 2: Record a trace of the "DisplayGreeting" sample](#)
- [Section 3: Analyze the trace file recording to identify the code issue](#)

Section 1: Build the sample code

In Section 1, you will build the sample code using Visual Studio.

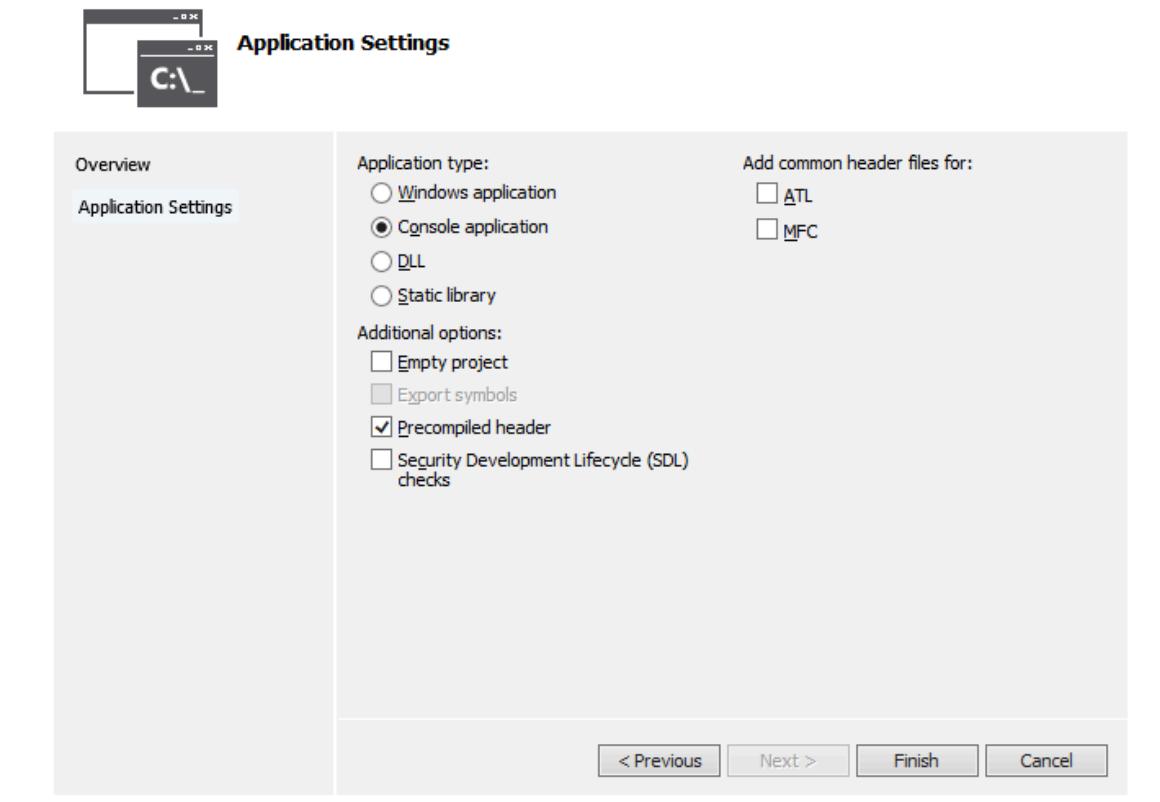
Create the sample app in Visual Studio

1. In Microsoft Visual Studio, click **File > New > Project/Solution...** and click on the **Visual C++ templates**.

Select the **Win32 Console Application**.

Provide a project name of **DisplayGreeting** and click on **OK**.

2. Uncheck the **Security Development Lifecycle (SDL) checks**.



3. Click on **Finish**.

4. Paste in the following text to the `DisplayGreeting.cpp` pane in Visual Studio.

```
C++
```

```
// DisplayGreeting.cpp : Defines the entry point for the console
application.
//



#include "stdafx.h"
#include <array>
#include <stdio.h>
#include <string.h>

void GetCppConGreeting(wchar_t* buffer, size_t size)
{
    wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD
LUCK IN ALL OF YOUR TIME TRAVEL DEBUGGING!";

    wcscpy_s(buffer, size, message);
}

int main()
{
    std::array <wchar_t, 50> greeting{};
    GetCppConGreeting(greeting.data(), sizeof(greeting));

    wprintf(L"%ls\n", greeting.data());
```

```
    return 0;  
}
```

5. In Visual Studio, click **Project > DisplayGreeting properties**. Then click on **C/C++** and **Code Generation**.

Set the following properties.

[+] Expand table

Setting	Value
Security Check	Disable Security Check (/GS-)
Basic Runtime Checks	Default

! **Note**

Although these setting are not recommended, it is possible to imagine a scenario where someone would advise using these settings to expedite coding or to facilitate certain testing environments.

6. In Visual Studio, click **Build > Build Solution**.

If all goes well, the build windows should display a message indicating that the build succeeded.

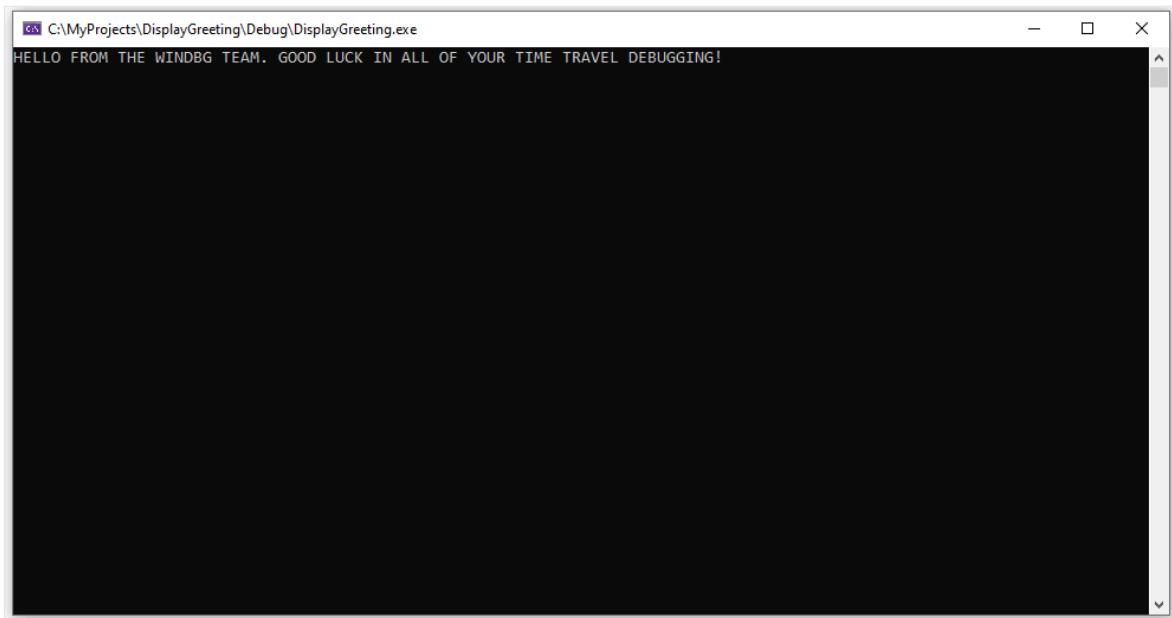
7. **Locate the built sample app files**

In the Solution Explorer, right click on the *DisplayGreeting* project and select **Open Folder in File explorer**.

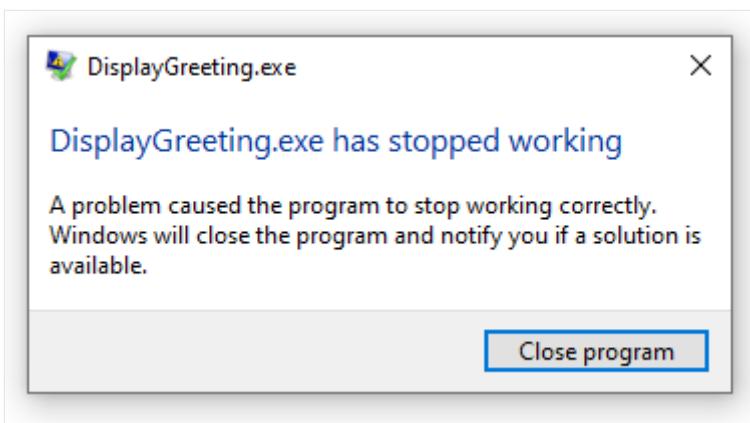
Navigate to the Debug folder that contains the complied exe and symbol pdb file for the sample. For example, you would navigate to *C:\Projects\DisplayGreeting\Debug*, if that's the folder that your projects are stored in.

8. **Run the sample app with the code flaw**

Double click on the exe file to run the sample app.



If this dialog box appears, select **Close program**



In the next section of the walkthrough, we will record the execution of the sample app to see if we can determine why this exception is occurring.

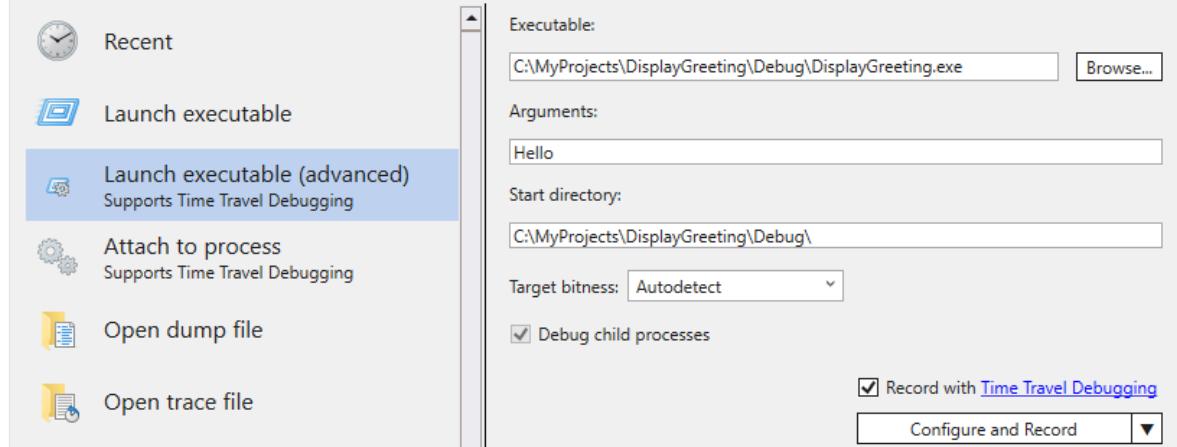
Section 2: Record a trace of the "DisplayGreeting" sample

In Section 2, you will record a trace of the misbehaving sample "DisplayGreeting" app

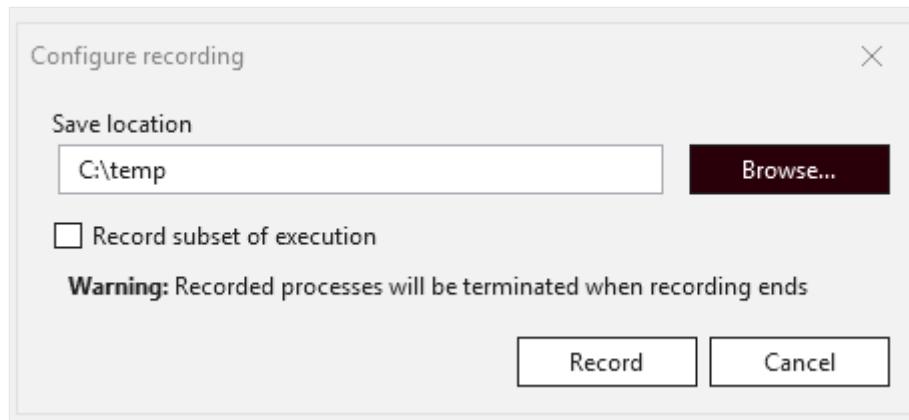
To launch the sample app and record a TTD trace, follow these steps. For general information about recording TTD traces, see [Time Travel Debugging - Record a trace](#)

1. Run WinDbg as an Administrator, so as to be able to record time travel traces.
2. In WinDbg, select **File > Start debugging > Launch executable (advanced)**.
3. Enter the path to the user mode executable that you wish to record or select **Browse** to navigate to the executable. For information about working with the launch executable menu in WinDbg, see [WinDbg - Start a user-mode session](#).

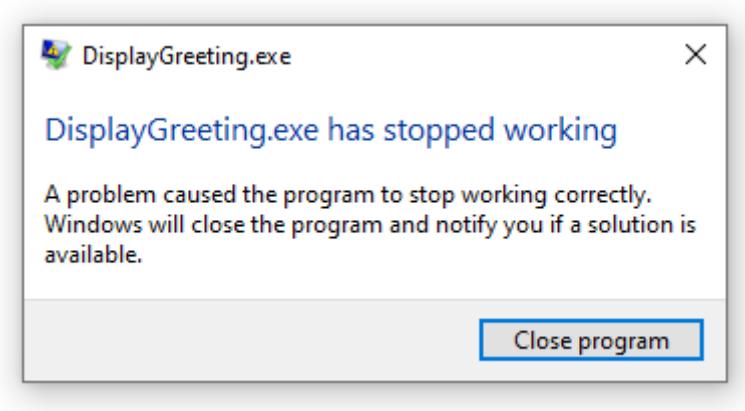
Start debugging



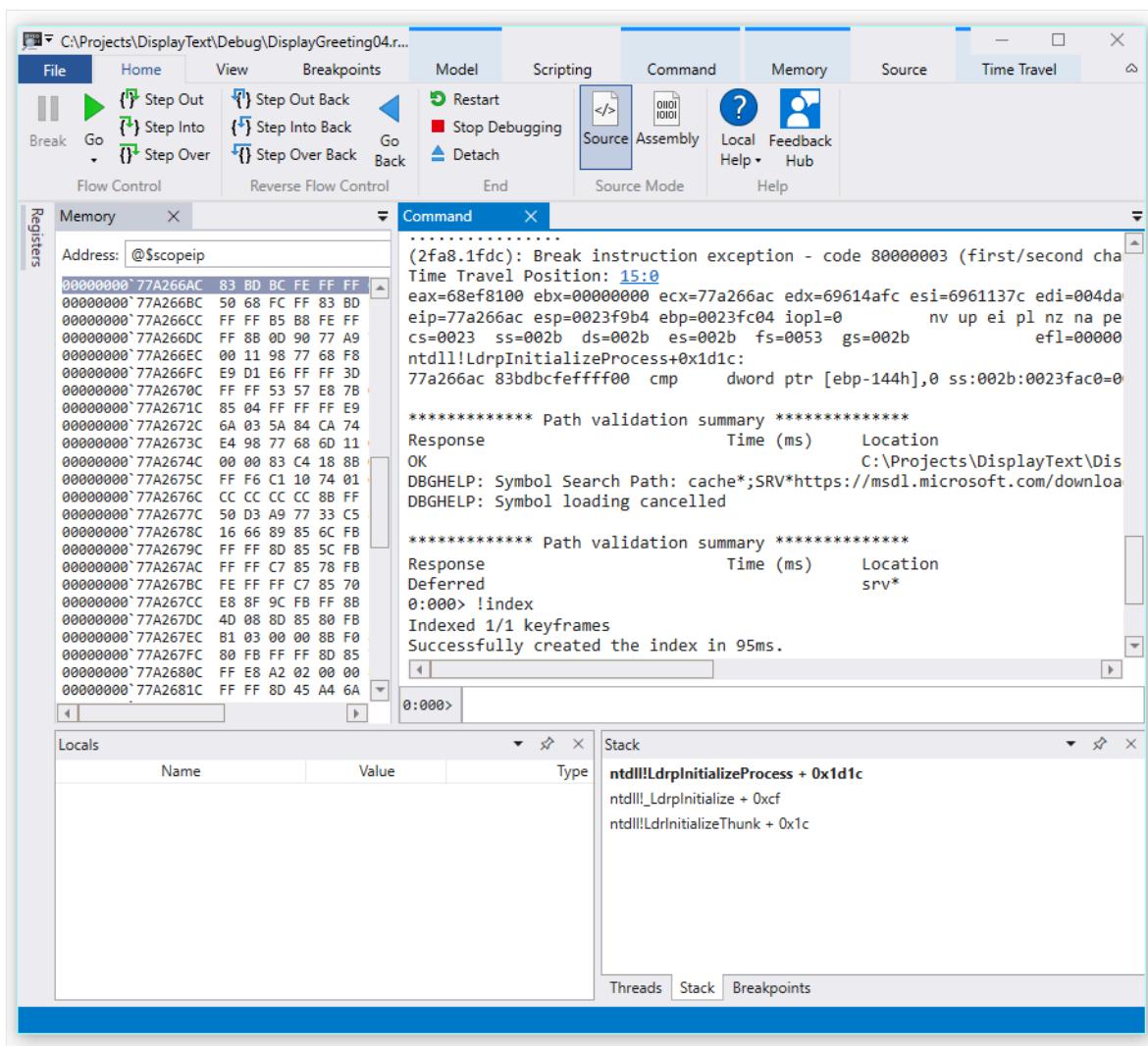
4. Check the **Record with Time Travel Debugging** box to record a trace when the executable is launched.
5. Click **Configure and Record** to start recording.
6. When the "Configure recording" dialog box appears, Click **Record** to launch the executable and start recording.



7. The recording dialog appears indicating the trace is being recorded. Shortly after that, the application crashes.
8. Click on **Close Program**, to dismiss the "DisplayGreeting has stopped working" dialog box.



9. When the program crashes, the trace file will be closed and written out to disk.



10. The debugger will automatically open the trace file and index it. Indexing is a process that enables efficient debugging of the trace file. This indexing process will take longer for larger trace files.

```
dbgcmd

(5120.2540): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: D:0 [Unindexed] Index
!index
```

```
Indexed 10/22 keyframes
Indexed 20/22 keyframes
Indexed 22/22 keyframes
Successfully created the index in 755ms.
```

ⓘ Note

A keyframe is a location in a trace used for indexing. Keyframes are generated automatically. Larger traces will contain more keyframes.

11. At this point you are at the beginning of the trace file and are ready to travel forward and backward in time.

Now that you have recorded a TTD trace, you can replay the trace back or work with the trace file, for example sharing it with a co-worker. For more information about working with trace files, see [Time Travel Debugging - Working with Trace Files](#)

In the next section of this lab we will analyze the trace file to locate the issue with our code.

Section 3: Analyze the trace file recording to identify the code issue

In Section 3, you will analyze the trace file recording to identify the code issue.

Configure the WinDbg Environment

1. Add your local symbol location to the symbol path and reload the symbols, by typing the following commands.

```
dbgcmd
.sympath+ C:\MyProjects\DisplayGreeting\Debug
.reload
```

2. Add your local code location to the source path by typing the following command.

```
dbgcmd
.srcpath+ C:\MyProjects\DisplayGreeting\DisplayGreeting
```

3. To be able to view the state of the stack and local variables, on the WinDbg ribbon, select **View** and **Locals** and **View** and **Stack**. Organize the windows to allow you to view them, the source code and the command windows at the same time.
4. On the WinDbg ribbon, select **Source** and **Open Source File**. Locate the `DisplayGreeting.cpp` file and open it.

Examine the exception

1. When the trace file was loaded it displays information that an exception occurred.

```
dbgcmd

2fa8.1fdc): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 15:0
eax=68ef8100 ebx=00000000 ecx=77a266ac edx=69614afc esi=6961137c
edi=004da000
eip=77a266ac esp=0023f9b4 ebp=0023fc04 iopl=0          nv up ei pl nz na
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp      dword ptr [ebp-144h],0
ss:002b:0023fac0=00000000
```

2. Use the `dx` command to list all of the events in the recording. The exception event is listed in the events.

```
dbgcmd

0:000> dx -r1 @$curprocess.TTD.Events
...
[0x2c]      : Module Loaded at position: 9967:0
[0x2d]      : Exception at 9BDC:0
[0x2e]      : Thread terminated at 9C43:0
...
```

⚠ Note

In this walkthrough three periods are used to indicate that extraneous output was removed.

3. Click on the Exception event to display information about that TTD event.

```
dbgcmd

0:000> dx -r1 @$curprocess.TTD.Events[17]
@$curprocess.TTD.Events[17] : Exception at 68:0
    Type      : Exception
    Position   : 68:0 [Time Travel]
    Exception  : Exception of type Hardware at PC: 0X540020
```

4. Click on the Exception field to further drill down on the exception data.

```
dbgcmd

0:000> dx -r1 @$curprocess.TTD.Events[17].Exception
@$curprocess.TTD.Events[17].Exception : Exception of
type Hardware at PC: 0X540020
    Position   : 68:0 [Time Travel]
    Type       : Hardware
    ProgramCounter : 0x540020
    Code        : 0xc0000005
    Flags       : 0x0
    RecordAddress : 0x0
```

The exception data indicates that this is a Hardware fault thrown by the CPU. It also provides the exception code of 0xc0000005 that indicates that this is an access violation. This typically indicates that we were attempting to write to memory that we don't have access to.

5. Click on the [Time Travel] link in the exception event to move to that position in the trace.

```
dbgcmd

0:000> dx @$curprocess.TTD.Events[17].Exception.Position.SeekTo()
Setting position: 68:0

@$curprocess.TTD.Events[17].Exception.Position.SeekTo()
(16c8.1f28): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 68:0
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046
edi=00191046
eip=00540020 esp=00effe4c ebp=00520055 iopl=0          nv up ei pl zr na
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
00540020 ??
```

Of note in this output is that the stack and base pointer are pointing to two very different addresses.

```
dbgcmd  
esp=00effe4c ebp=00520055
```

This could indicate that stack corruption - possibly a function returned and then corrupted the stack. To validate this, we need to travel back to before the CPU state was corrupted and see if we can determine when the stack corruption occurred.

Examine the local variables and set a code breakpoint

At the point of failure in trace it is common to end up a few steps after the true cause in error handling code. With time travel we can go back an instruction at a time, to locate the true root cause.

1. From the **Home** ribbon use the **Step Into Back** command to step back three instructions. As you do this, continue to examine the stack and memory windows.

The command window will display the time travel position and the registers as you step back three instructions.

```
dbgcmd  
  
0:000> t-  
Time Travel Position: 67:40  
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046  
edi=00191046  
eip=00540020 esp=00effe4c ebp=00520055 iopl=0 nv up ei pl zr na  
pe nc  
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b  
efl=00000246  
00540020 ?? ???  
  
0:000> t-  
Time Travel Position: 67:3F  
eax=00000000 ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046  
edi=00191046  
eip=0019193d esp=00effe48 ebp=00520055 iopl=0 nv up ei pl zr na  
pe nc  
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b  
efl=00000246  
DisplayGreeting!main+0x4d:  
0019193d c3  
  
0:000> t-  
Time Travel Position: 67:39
```

```
eax=0000004c ebx=00cf8000 ecx=99da9203 edx=69cf1a6c esi=00191046
edi=00191046
eip=00191935 esp=00efffd94 ebp=00effe44 iopl=0          nv up ei pl nz ac
po nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
efl=00000212
DisplayGreeting!main+0x45:
```

ⓘ Note

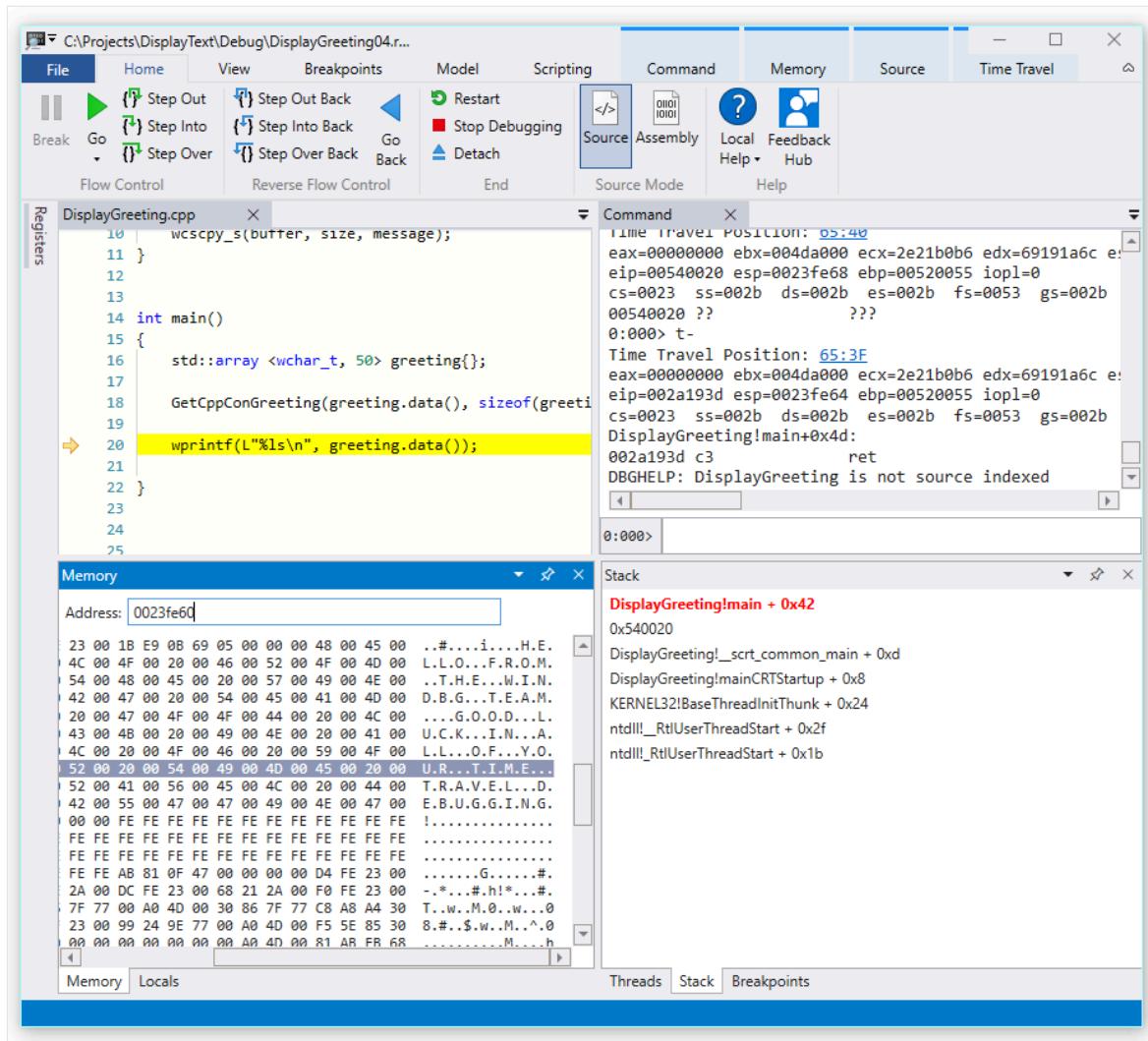
In this walkthrough, the command output shows the commands that can be used instead of the UI Menu options to allow users with a command line usage preference to use command line commands.

2. At this point in the trace our stack and base pointer have values that make more sense, so it appears that we have getting closer to the point in the code where the corruption occurred.

```
dbgcmd
esp=00efffd94 ebp=00effe44
```

Also of interest is that the locals window contains values from our target app and the source code window is highlighting the line of code that is ready to be executed at this point in the trace.

3. To further investigate, we can open up a memory window to view the contents near the base pointer memory address of *0x00effe44*.
4. To display the associated ASCII characters, from the Memory ribbon, select **Text** and then **ASCII**.



5. Instead of the base pointer pointing to an instruction it is pointing to our message text. So something is not right here, this may be close to the point in time that we have corrupted the stack. To further investigate we will set a breakpoint.

ⓘ Note

In this very small sample it would be pretty easy to just look in the code, but if there are hundreds of lines of code and dozens of subroutines the techniques described here can be used to decrease the time necessary to locate the issue.

TTD and breakpoints

Using breakpoints is a common approach to pause code execution at some event of interest. TTD allows you to set a breakpoint and travel back in time until that breakpoint is hit after the trace has been recorded. The ability to examine the process state after an issue has happened, to determine the best location for a breakpoint, enables additional debugging workflows unique to TTD.

Memory access breakpoints

You can set breakpoints that fire when a memory location is accessed. Use the **ba** (break on access) command, with the following syntax.

```
dbgcmd  
ba <access> <size> <address> {options}
```

[\[+\] Expand table](#)

Option	Description
e	execute (when CPU fetches an instruction from the address)
r	read/write (when CPU reads or writes to the address)
w	write (when the CPU writes to the address)

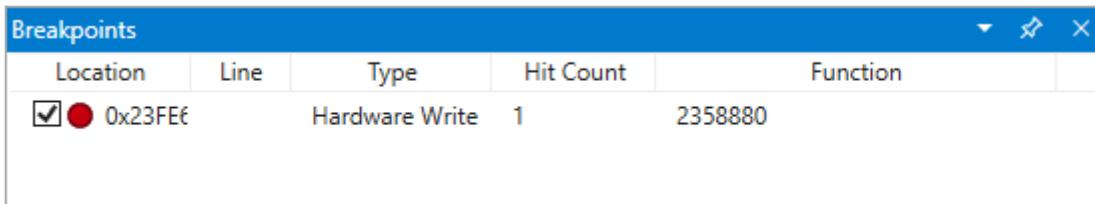
Note that you can only set four data breakpoints at any given time and it is up to you to make sure that you are aligning your data correctly or you won't trigger the breakpoint (words must end in addresses divisible by 2, dwords must be divisible by 4, and quadwords by 0 or 8).

Set the break on memory access breakpoint for the base pointer

1. At this point in the trace we would like to set a breakpoint on write memory access to base pointer - ebp which in our example is 00effe44. To do this use the **ba** command using the address we want to monitor. We want to monitor writes for four bytes, so we specify w4.

```
dbgcmd  
0:000> ba w4 00effe44
```

2. Select **View** and then **Breakpoints** to confirm they are set as intended.



3. From the Home menu, select **Go Back** to travel back in time until the breakpoint is hit.

```
dbgcmd
```

```

0:000> g-
Breakpoint 0 hit
Time Travel Position: 5B:92
eax=0000000f ebx=003db000 ecx=00000000 edx=00cc1a6c esi=00d41046
edi=0053fde8
eip=00d4174a esp=0053fcf8 ebp=0053fde8 iopl=0          nv up ei pl nz ac
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000216
DisplayGreeting!DisplayGreeting+0x3a:
00d4174a c745e0000000000 mov      dword ptr [ebp-20h],0
ss:002b:0053fdc8=cccccccc

```

4. Select **View** and then **Locals**. In the locals window we can see that the *destination* variable has only part of the message, while the *source* has contains all of the text. This information supports the idea that the stack was corrupted.

Locals			
Name	Value		
available	0x31		unsigned
destination_it	0xa4fefe : "R\1???"		wchar_t ^
source_it	0xb66b96 : "R TIME TRAVEL DEBUGGING!"		wchar_t ^
destination	0xa4fe98 : "HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR\1??"		wchar_t ^
size_in_elements	0x64		unsigned
source	0xb66b30 : "HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR TIME TRAVEL		wchar_t ^

5. At this point we can examine the program stack to see what code is active. From the **View** ribbon select **Stack**.

Stack
ucrtbased!common_tcscpy_s<wchar_t> + 0x187
ucrtbased!wcscpy_s + 0x16
DisplayGreeting!GetCppConGreeting + 0x22
DisplayGreeting!main + 0x2c
DisplayGreeting!invoke_main + 0x1e
DisplayGreeting!_scrt_common_main_seh + 0x15a
DisplayGreeting!_scrt_common_main + 0xd
DisplayGreeting!mainCRTStartup + 0x8
KERNEL32!BaseThreadInitThunk + 0x24

As it is very unlikely that the Microsoft provided wcscpy_s() function would have a code bug like this, we look further in the stack. The stack shows that Greeting!main calls Greeting!GetCppConGreeting. In our very small code sample we could just open the code at this point and likely find the error pretty easily. But to illustrate the techniques that can be used with larger, more complex program, we will set a new breakpoint to investigate further.

Set the break on access breakpoint for the GetCppConGreeting function

1. Use the breakpoints window to clear the existing breakpoint by right clicking on the existing breakpoint and selecting **Remove**.

2. Determine the address of the `DisplayGreeting!GetCppConGreeting` function using the `dx` command.

```
dbgcmd

0:000> dx &DisplayGreeting!GetCppConGreeting
&DisplayGreeting!GetCppConGreeting : 0xb61720 [Type:
void (__cdecl*)(wchar_t *,unsigned int)]
[Type: void __cdecl(wchar_t *,unsigned int)]
```

3. Use the `ba` command to set a breakpoint on memory access. Because the function will just be read from memory for execution, we need to set a `r` - read breakpoint.

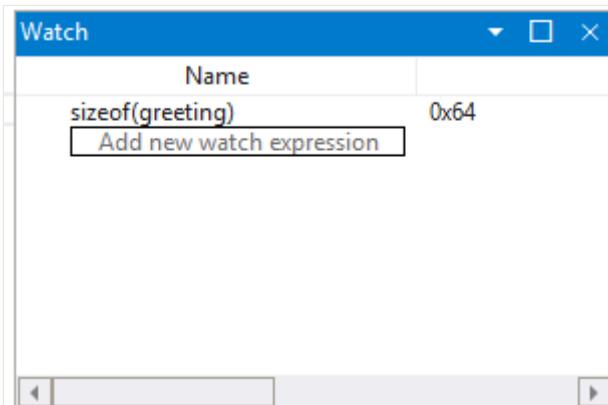
```
dbgcmd

0:000> ba r4 b61720
```

4. Confirm that a Hardware Read breakpoint is active in the breakpoints window.

Breakpoints					
Location	Line	Type	Hit Count	Function	
<input checked="" type="checkbox"/> c:\projects\displaygreeting\displaygreeting\displaygreeting.cpp	7	Hardware Read	1	11933472	

5. As we are wondering about the size of the greeting string we will set a watch window to display the value of `sizeof(greeting)`. From the View ribbon, select **Watch** and provide `sizeof(greeting)`. If the value is not in scope the watch window will display - *Unable to bind name 'greeting'*.



6. On the Time Travel menu, use **Time travel to start** or use the `!tt 0` command to move to the start of the trace.

```
dbgcmd

0:000> !tt 0
Setting position to the beginning of the trace
```

```
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c
edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0           nv up ei pl nz na
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp     dword ptr [ebp-144h],0
ss:002b:00ddf4c4=00000000
```

7. On the Home menu, select **Go** or use the `g` command, to move forward in the code until the breakpoint is hit.

```
dbgcmd

0:000> g
Breakpoint 2 hit
Time Travel Position: 4B:1AD
eax=00ddf800 ebx=00fa2000 ecx=00ddf800 edx=00b61046 esi=00b61046
edi=00b61046
eip=00b61721 esp=00ddf7a4 ebp=00ddf864 iopl=0           nv up ei pl nz na
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
DisplayGreeting!GetCppConGreeting+0x1:
00b61721 8bec          mov     ebp,esp
```

8. On the Home menu, select **Step Out Back** or use the `g-u` command to back one step.

```
dbgcmd

0:000> g-u
Time Travel Position: 4B:1AA
eax=00ddf800 ebx=00fa2000 ecx=00ddf800 edx=00b61046 esi=00b61046
edi=00b61046
eip=00b61917 esp=00ddf7ac ebp=00ddf864 iopl=0           nv up ei pl nz na
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
DisplayGreeting!main+0x27:
00b61917 e8def7ffff    call    DisplayGreeting!ILT+245(?GetCppConGreetingYAXPA_WIZ) (00b610fa)
```

9. It looks like we have found the root cause. The *greeting* array that we declared is 50 characters in length, while the `sizeof(greeting)` that we pass into `GetCppConGreeting` is 0x64, 100.

The screenshot shows the WinDbg debugger interface. On the left, there is assembly code for a C++ program. Lines 16 and 18 are highlighted in yellow, indicating they are currently being executed or are part of the current stack frame. Line 18 contains the call to `GetCppConGreeting`. On the right, a 'Watch' window is open, showing a single entry: `sizeof(greeting)` with the value `0x64`.

```
6 void GetCppConGreeting(wchar_t* buffer, size_t size)
7 {
8     wchar_t const* const message = L"HELLO FROM THE WINDBG TEAM. GOOD LUCK
9
10    wcscpy_s(buffer, size, message);
11 }
12
13
14 int main()
15 {
16     std::array <wchar_t, 50> greeting{};
17
18     GetCppConGreeting(greeting.data(), sizeof(greeting));
19
20     wprintf(L"%ls\n", greeting.data());
21
22 }
```

Name	Value
sizeof(greeting)	0x64

As we look at the size issue further, we also notice that the message is 75 characters in length, and is 76 when including the end of string character.

dbgcmd

HELLO FROM THE WINDBG TEAM. GOOD LUCK IN ALL OF YOUR TIME TRAVEL
DEBUGGING!

10. One way to fix the code would be to expand the size of the character array to 100.

C++

```
std::array <wchar_t, 100> greeting{};
```

And we also need to change `sizeof(greeting)` to `size(greeting)` in this line of code.

C++

```
GetCppConGreeting(greeting.data(), size(greeting));
```

11. To validate these fixes, we could recompile the code and confirm that it runs without error.

Setting a breakpoint using the source window

1. An alternative way to perform this investigation would be to set a breakpoint by clicking on any line of code. For example clicking on the right side of the std::array definition line in the source window will set a breakpoint there.

The screenshot shows the Microsoft Visual Studio Source Editor. A red circle highlights the line number 16 in the left margin, indicating where a breakpoint was set. The code editor displays the following C++ code:

```
4 #include <string.h>
5
6 void GetCppConGreeting(wchar_t* buffer, size_t size)
7 {
8     wchar_t const* const message = L"HELLO FROM THE WINDBG TEA
9
10    wcscpy_s(buffer, size, message);
11 }
12
13
14 int main()
15 {
16     std::array<wchar_t, 50> greeting{};
17
18     GetCppConGreeting(greeting.data(), sizeof(greeting));
19
20     wprintf(L"%ls\n", greeting.data());
21
22 }
```

Below the code editor is the "Breakpoints" window, which lists a single breakpoint:

Location	Line
c:\projects\displaygreeting\displaygreeting\displaygreeting.cpp	16

2. On the Time Travel menu, use **Time travel to start** command to move to the start of the trace.

```
dbgcmd

0:000> !tt 0
Setting position to the beginning of the trace
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c
```

```
edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0          nv up ei pl nz na
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfeffff00  cmp      dword ptr [ebp-144h],0
ss:002b:00ddf4c4=00000000
```

3. On the Home Ribbon click on **Go** to travel back until the breakpoint is hit.

```
dbgcmd

Breakpoint 0 hit
Time Travel Position: 5B:AF
eax=0000000f ebx=00c20000 ecx=00000000 edx=00000000 esi=013a1046
edi=00efffa60
eip=013a17c1 esp=00eff970 ebp=00efffa60 iopl=0          nv up ei pl nz na
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
DisplayGreeting!DisplayGreeting+0x41:
013a17c1 8bf4        mov      esi,esp
```

Set the break on access breakpoint for the *greeting* variable

Another alternative way to perform this investigation, would be to set a breakpoint on suspect variables and examine what code is changing them. For example, to set a breakpoint on the greeting variable in the GetCppConGreeting method, use this procedure.

This portion of the walkthrough assumes that you are still located at the breakpoint from the previous section.

1. From **View** and then **Locals**. In the locals window, *greeting* is available in the current context, so we will be able to determine its memory location.
2. Use the **dx** command to examine the *greeting* array.

```
dbgcmd

0:000> dx &greeting
&greeting           : ddf800 : { size=50 } [Type:
std::array<wchar_t,50> *]
[<Raw View>]      [Type: std::array<wchar_t,50>]
[0]                 : 3 [Type: wchar_t]
[1]                 : 0 [Type: wchar_t]
```

In this trace, *greeting* is located in memory at ddf800.

3. Use the breakpoints window to clear any existing breakpoint by right clicking on the existing breakpoint and selecting **Remove**.
4. Set the breakpoint with the **ba** command using the memory address we want to monitor for write access.

```
dbgcmd
```

```
ba w4 ddf800
```

5. On the Time Travel menu, use **Time travel to start** command to move to the start of the trace.

```
dbgcmd
```

```
0:000> !tt 0
Setting position to the beginning of the trace
Setting position: 15:0
(1e5c.710): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 15:0
eax=68e28100 ebx=00000000 ecx=77a266ac edx=69e34afc esi=69e3137c
edi=00fa2000
eip=77a266ac esp=00ddf3b8 ebp=00ddf608 iopl=0          nv up ei pl nz na
pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000206
ntdll!LdrpInitializeProcess+0x1d1c:
77a266ac 83bdbcfffff00  cmp      dword ptr [ebp-144h],0
ss:002b:00ddf4c4=00000000
```

6. On the Home menu, select **Go** to travel forward to the first point of memory access of the greeting array.

```
dbgcmd
```

```
0:000> g-
Breakpoint 0 hit
Time Travel Position: 5B:9C
eax=cccccccc ebx=002b1000 ecx=00000000 edx=68d51a6c esi=013a1046
edi=001bf7d8
eip=013a1735 esp=001bf6b8 ebp=001bf7d8 iopl=0          nv up ei pl nz na
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
DisplayGreeting!GetCppConGreeting+0x25:
```

```
013a1735 c745ec04000000    mov        dword ptr [ebp-14h],4  
ss:002b:001bf7c4=cccccccc
```

Alternatively, we could have traveled to the end of the trace and worked in reverse through the code to find that last point in the trace that the array memory location was written to.

Use the TTD.Memory objects to view memory access

Another way to determine at what points in the trace memory has been accessed, is to use the TTD.Memory objects and the dx command.

1. Use the `dx` command to examine the *greeting* array.

```
0:000> dx &greeting  
&greeting : 0xddf800 [Type: std::array<wchar_t,50> *]  
[+0x000] _Elems : "詠棟櫛鬲???" [Type: wchar_t [50]]
```

In this trace, *greeting* is located in memory at ddf800.

2. Use the **dx** command to look at the four bytes in memory starting at that address with the read write access.

```
dbgcmd

0:000> dx -r1 @$cursession.TTD.Memory(0xdddf800,0xdddf804, "rw")
@$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
[0x0]
[0x1]
[0x2]
[0x3]
[0x4]
[0x5]
[0x6]
[0x7]
[0x8]
[0x9]
[0xa]
...

```

3. Click on any of the occurrences to display more information about that occurrence of memory access.

dhacmd

```

0:000> dx -r1 @$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5]
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5]
    EventType      : MemoryAccess
    ThreadId       : 0x710
    UniqueThreadId : 0x2
    TimeStart      : 27:3C1 [Time Travel]
    TimeEnd        : 27:3C1 [Time Travel]
    AccessType     : Write
    IP             : 0x6900432f
    Address        : 0xddf800
    Size           : 0x4
    Value          : 0xddf818
    OverwrittenValue : 0x0
    SystemTimeStart : Monday, November 18, 2024 23:01:43.400
    SystemTimeEnd   : Monday, November 18, 2024 23:01:43.400

```

4. Click on [Time Travel] for TimeStart to position the trace at the point in time.

```

dbgcmd

0:000> dx @$cursession.TTD.Memory(0xddf800,0xddf804, "rw")
[5].TimeStart.SeekTo()
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw")[5].TimeStart.SeekTo()
(1e5c.710): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 27:3C1
eax=00ddf81c ebx=00fa2000 ecx=00ddf818 edx=ffffffff esi=00000000
edi=00b61046
eip=6900432f esp=00ddf804 ebp=00ddf810 iopl=0          nv up ei pl nz ac
po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000212
ucrtbased!_register_onexit_function+0xf:
6900432f 51          push    ecx

```

5. If we are interested in the last occurrence of read/write memory access in the trace we can click on the last item in the list or append the .Last() function to the end of the dx command.

```

dbgcmd

0:000> dx -r1 @$cursession.TTD.Memory(0xddf800,0xddf804, "rw").Last()
@$cursession.TTD.Memory(0xddf800,0xddf804, "rw").Last()
    EventType      : MemoryAccess
    ThreadId       : 0x710
    UniqueThreadId : 0x2
    TimeStart      : 53:100E [Time Travel]
    TimeEnd        : 53:100E [Time Travel]
    AccessType     : Read
    IP             : 0x690338e4

```

Address	:	0xddf802
Size	:	0x2
Value	:	0x45
SystemTimeStart	:	Monday, November 18, 2024 23:01:43.859
SystemTimeEnd	:	Monday, November 18, 2024 23:01:43.859

6. We could then click on [Time Travel] to move to that position in the trace and look further at the code execution at that point, using the techniques described earlier in this lab.

For more information about the TTD.Memory objects, see [TTD.Memory Object](#).

Summary

In this very small sample the issue could have been determined by looking at the few lines of code, but in larger programs the techniques presented here can be used to decrease the time necessary to locate an issue.

Once a trace is recorded, the trace and repro steps can be shared, and the issue will be reproducible on any PC.

See Also

[Time Travel Debugging - Overview](#)

[Time Travel Debugging - Recording](#)

[Time Travel Debugging - Replay a trace](#)

[Time Travel Debugging - Working with trace files](#)

Feedback

Was this page helpful?

 Yes

 No

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

Introduction to Time Travel Debugging objects

Article • 11/20/2024



This section describes how to use the data model to query time travel traces. This can be a useful tool to answer questions like these about the code that is captured in a time travel trace.

- What exceptions are in the trace?
- At what point in time in the trace did a specific code module load?
- When were threads created/terminated in the trace?
- What are the longest running threads in the trace?

There are TTD extensions that add data to the *Session* and *Process* data model objects. The TTD data model objects can be accessed through the [dx \(Display Debugger Object Model Expression\)](#) command, WinDbg's model windows, JavaScript and C++. The TTD extensions are automatically loaded when debugging a time travel trace.

Process Objects

The primary objects added to *Process* objects can be found in the *TTD* namespace off of any *Process* object. For example, `@$curprocess.TTD`.

```
dbgcmd

0:000> dx @$curprocess.TTD
@$curprocess.TTD
  Index
  Threads
  Events
  DebugOutput
    Lifetime      : [2C8:0, 16EC:98A]
    DefaultMemoryPolicy : InFragmentAggressive
    SetPosition     [Sets the debugger to point to the given position on
this process.]
    GatherMemoryUse [0]
    RecordClients
    PrevMemoryAccess [(accessMask, address, size [, address, size, ...]) -
Find the previous matching memory access before current position.]
    NextMemoryAccess [(accessMask, address, size [, address, size, ...]) -
```

Find the next matching memory access after current position.]
Recorder

For general information on working with LINQ queries and debugger objects, see [Using LINQ With the debugger objects](#).

Properties

[+] Expand table

Object	Description
Lifetime	A TTD range object describing the lifetime of the entire trace.
Threads	Contains a collection of TTD thread objects , one for every thread throughout the lifetime of the trace.
Events	Contains a collection of TTD event objects , one for every event in the trace.

Methods

[+] Expand table

Method	Description
SetPosition()	Takes an integer between 0 and 100 or string in N:N form as input and jumps the trace to that location. See !tt for more information.

Session Objects

The primary objects added to *Session* objects can be found in the *TTD* namespace off of any *Session* object. For example, `@$cursession.TTD`.

dbgcmd

```
0:000> dx @$cursession.TTD
@$cursession.TTD
    Calls          [Returns call information from the trace for the
                    specified set of methods: TTD.Calls("module!method1", "module!method2", ...)]
    For example: dx @$cursession.TTD.Calls("user32!SendMessageA")
    Memory         [Returns memory access information for specified
                    address range: TTD.Memory(startAddress, endAddress [, "rwec"])]
    MemoryForPositionRange [Returns memory access information for specified
                           address range and position range: TTD.MemoryForPositionRange(startAddress,
                           endAddress [, "rwec"], minPosition, maxPosition)]
```

```

PinObjectPosition [Pins an object to the given time position:
TTD.PinObjectPosition(obj, pos)]
    AsyncQueryEnabled : false
    RichQueryTypesEnabled : true
    DefaultParameterCount : 0x4
    Data : Normalized data sources based on the contents of the
time travel trace
    Utility : Methods that can be useful when analyzing time travel
traces
    Analyzers : Methods that perform code analysis on the time travel
trace
    Bookmarks : Bookmark collection
    Checkers : Checkers (scripts for detection of common issues
recorded in a time travel trace)

```

ⓘ Note

There are some objects and methods added by TTDAnalyze that are used for internal functions of the extension. Not all namespaces are documented, and the current namespaces will evolve over time.

Methods

[] Expand table

Method	Description
Data.Heap()	A collection of heap objects that were allocated during the trace. Note that this is a function that does computation, so it takes a while to run.
Calls()	Returns a collection of calls objects that match the input string. The input string can contain wildcards. Note that this is a function that does computation, so it takes a while to run.
Memory()	This is a method that takes beginAddress, endAddress and dataAccessMask parameters and returns a collection of memory objects . Note that this is a function that does computation, so it takes a while to run.

Sorting query output

Use the OrderBy() method to sort the rows returned from the query by one or more columns. This example sorts by TimeStart in ascending order.

dbgcmd

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").OrderBy(c =>
c.TimeStart)
@$cursession.TTD.Calls("kernelbase!GetLastError").OrderBy(c => c.TimeStart)
[0x0]
    EventType      : 0x0
    ThreadId       : 0x2d98
    UniqueThreadId : 0x2
    TimeStart       : 718:7D7 [Time Travel]
    TimeEnd         : 718:7DA [Time Travel]
    Function        : KERNELBASE!GetLastError
    FunctionAddress : 0x7ff996cf20f0
    ReturnAddress   : 0x7ff99855ac5a
    ReturnValue     : 0x0 [Type: unsigned long]
    Parameters
    SystemTimeStart : Friday, January 12, 2024 21:18:40.862
    SystemTimeEnd   : Friday, January 12, 2024 21:18:40.862
[0x1]
    EventType      : 0x0
    ThreadId       : 0x2d98
    UniqueThreadId : 0x2
    TimeStart       : 72D:1B3 [Time Travel]
    TimeEnd         : 72D:1B6 [Time Travel]
    Function        : KERNELBASE!GetLastError
    FunctionAddress : 0x7ff996cf20f0
    ReturnAddress   : 0x7ff9961538df
    ReturnValue     : 0x57 [Type: unsigned long]
    Parameters
    SystemTimeStart : Friday, January 12, 2024 21:18:40.862
    SystemTimeEnd   : Friday, January 12, 2024 21:18:40.862
...
...
```

To display additional depth of the data model objects the `-r2` recursion level option is used. For more information about the `dx` command options, see [dx \(Display Debugger Object Model Expression\)](#).

This example sorts by `TimeStart` in descending order.

```
dbgcmd

0:000> dx -r2
@$cursession.TTD.Calls("kernelbase!GetLastError").OrderByDescending(c =>
c.TimeStart)
@$cursession.TTD.Calls("kernelbase!GetLastError").OrderByDescending(c =>
c.TimeStart)
[0x1896]
    EventType      : Call
    ThreadId       : 0x3a10
    UniqueThreadId : 0x2
    TimeStart       : 464224:34 [Time Travel]
    TimeEnd         : 464224:37 [Time Travel]
    Function        : UnknownOrMissingSymbols
```

```
FunctionAddress : 0x7561ccc0
ReturnAddress   : 0x7594781c
ReturnValue     : 0x0
Parameters
[0x18a0]
  EventType      : Call
  ThreadId       : 0x3a10
  UniqueThreadId : 0x2
  TimeStart      : 464223:21 [Time Travel]
  TimeEnd        : 464223:24 [Time Travel]
  Function        : UnknownOrMissingSymbols
  FunctionAddress : 0x7561ccc0
  ReturnAddress   : 0x7594781c
  ReturnValue     : 0x0
  Parameters
```

Specifying elements in a query

To select a specific element a variety of qualifiers can be appended to the query. For example, the query displays the first call that contains "kernelbase!GetLastError".

```
dbgcmd

0:000> dx @$cursession.TTD.Calls("kernelbase!GetLastError").First()
@$cursession.TTD.Calls("kernelbase!GetLastError").First()
  EventType      : 0x0
  ThreadId       : 0x2d98
  UniqueThreadId : 0x2
  TimeStart      : 718:7D7 [Time Travel]
  TimeEnd        : 718:7DA [Time Travel]
  Function        : KERNELBASE!GetLastError
  FunctionAddress : 0x7ff996cf20f0
  ReturnAddress   : 0x7ff99855ac5a
  ReturnValue     : 0x0 [Type: unsigned long]
  Parameters
  SystemTimeStart : Friday, January 12, 2024 21:18:40.862
  SystemTimeEnd   : Friday, January 12, 2024 21:18:40.862
```

Filtering in a query

Use the `Select()` method to choose which columns to see and modify the column display name.

This example returns rows where `ReturnValue` is not zero and selects to display the `TimeStart` and `ReturnValue` columns with custom display names of `Time` and `Error`.

```
dbgcmd
```

```
0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c =>
c.ReturnValue != 0).Select(c => new { Time = c.TimeStart, Error =
c.ReturnValue })
@$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue
!= 0).Select(c => new { Time = c.TimeStart, Error = c.ReturnValue })
[0x1]
    Time          : 72D:1B3 [Time Travel]
    Error         : 0x57 [Type: unsigned long]
[0x2]
    Time          : 72D:1FC [Time Travel]
    Error         : 0x2af9 [Type: unsigned long]
[0x3]
    Time          : 72D:26E [Time Travel]
    Error         : 0x2af9 [Type: unsigned long]
```

Grouping

Use the GroupBy() method to group data returned by the query to do perform analysis using structured results This example groups the time locations by error number.

```
dbgcmd

0:000> dx -r2 @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c =>
c.ReturnValue != 0).Select(c => new { Time = c.TimeStart, Error =
c.ReturnValue }).GroupBy(x => x.Error)
@$s = @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c =>
c.ReturnValue != 0).Select(c => new { Time = c.TimeStart, Error =
c.ReturnValue }).GroupBy(x => x.Error)
[0x36b7]
[0x0]
[0x1]
[0x2]
[0x3]
[...]
[0x3f0]
[0x0]
[0x1]
[0x2]
[0x3]
...
...
```

Assigning result of a query to a variable

Use this syntax to assigning result of a query to a variable `dx @$var = <expression>`

This example assigns the results of a query to myResults

```
dbgcmd
```

```
dx -r2 @$myResults =
@$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue
!= 0).Select(c => new { Time = c.TimeStart, Error = c.ReturnValue })
```

Use the dx command to display the newly created variable using the -g grid option. For more information on the dx command options, see [dx \(Display Debugger Object Model Expression\)](#).

```
dbgcmd
```

```
0:000> dx -g @$myResults
=====
=      = (+) Time      = (+) Error =
=====
= [0x13]  - 3C64A:834  - 0x36b7    =
= [0x1c]  - 3B3E7:D6   - 0x3f0     =
= [0x1d]  - 3C666:857  - 0x36b7    =
= [0x20]  - 3C67E:12D   - 0x2      =
= [0x21]  - 3C6F1:127  - 0x2      =
= [0x23]  - 3A547:D6   - 0x3f0    =
= [0x24]  - 3A59B:D0   - 0x3f0    =
```

Examples

Querying for exceptions

This LINQ query uses the [TTD.Event object](#) to display all of the exceptions in the trace.

```
dbgcmd
```

```
0:000> dx @$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e
=> e.Exception)
@$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e =>
e.Exception)
[0x0]          : Exception 0x000006BA of type Software at PC:
0X777F51D0
[0x1]          : Exception 0x000006BA of type Software at PC:
0X777F51D0
[0x2]          : Exception 0xE06D7363 of type CPlusPlus at PC:
0X777F51D0
```

Querying for specific API calls

Use [TTD.Calls object](#) to query for specific API calls. In this example, an error has occurred when calling `user32!MessageBoxW`, the Windows API to show a message box. We list all calls to `MessageBoxW`, order it by the start time of the function, and then pick the last call.

```
dbgcmd

0:000> dx @$cursession.TTD.Calls("user32!MessageBoxW").OrderBy(c =>
c.TimeStart).Last()
@$cursession.TTD.Calls("user32!MessageBoxW").OrderBy(c =>
c.TimeStart).Last()
  EventType      : Call
  ThreadId       : 0x3a10
  UniqueThreadId : 0x2
  TimeStart      : 458310:539 [Time Travel]
  TimeEnd        : 45C648:61 [Time Travel]
  Function       : UnknownOrMissingSymbols
  FunctionAddress : 0x750823a0
  ReturnAddress   : 0x40cb93
  ReturnValue     : 0x10a7000000000001
  Parameters
```

Querying for the load event of a specific module

First, use the [!m \(List Loaded Modules\)](#) command to display the loaded modules.

```
dbgcmd

0:000> !m
start    end      module name
012b0000 012cf000 CDog_Console  (deferred)
11570000 1158c000 VCRUNTIME140D (deferred)
11860000 119d1000 ucrtbased   (deferred)
119e0000 11b63000 TTDRecordCPU (deferred)
11b70000 11cb1000 TTDWriter    (deferred)
73770000 73803000 apphelp     (deferred)
73ea0000 74062000 KERNELBASE   (deferred)
75900000 759d0000 KERNEL32    (deferred)
77070000 771fe000 ntdll       (private pdb symbols)
```

Then use the following dx command to see at what position in the trace a specific module was loaded, such as `ntdll`.

```
dbgcmd

dx @$curprocess.TTD.Events.Where(t => t.Type == "ModuleLoaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
```

```
@$curprocess.TTD.Events.Where(t => t.Type == "ModuleLoaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
[0x0] : Module Loaded at position: A:0
```

This LINQ query displays the load event(s) of a particular module.

```
dbgcmd

0:000> dx @$curprocess.TTD.Events.Where(t => t.Type ==
"ModuleUnloaded").Where(t => t.Module.Name.Contains("ntdll.dll"))
@$curprocess.TTD.Events.Where(t => t.Type == "ModuleUnloaded").Where(t =>
t.Module.Name.Contains("ntdll.dll"))
[0x0] : Module Unloaded at position: FFFFFFFFFFFFFFE:0
```

The address of FFFFFFFFFFFFFFE:0 indicates the end of the trace.

Querying for all of the error checks in the trace

Use this command to sort by all of the error checks in the trace by error count.

```
dbgcmd

0:000> dx -g @$cursession.TTD.Calls("kernelbase!GetLastError").Where( x=>
x.ReturnValue != 0).GroupBy(x => x.ReturnValue).Select(x => new {
ErrorNumber = x.First().ReturnValue, ErrorCount =
x.Count()}).OrderByDescending(p => p.ErrorCount),d
=====
= (+) ErrorNumber = ErrorCount =
=====
= [1008]      - 1008          - 8668      =
= [14007]     - 14007         - 4304      =
= [2]          - 2            - 1710      =
= [6]          - 6            - 1151      =
= [1400]      - 1400         - 385       =
= [87]         - 87           - 383       =
```

Querying for the time position in the trace when threads were created

Use this dx command to display all of the events in the trace in grid format (-g).

```
dbgcmd

0:000> dx -g @$curprocess.TTD.Events
=====
```

```
= = (+) Type
= (+) Position      = (+) Module
= (+) Thread       =
=====
=====
=====
= [0x0] : Module Loaded at position: 2:0          - ModuleLoaded
- 2:0           - Module C:\Users\USER1\Documents\Visual Studio
2015\Proj... -
= [0x1] : Module Loaded at position: 3:0          - ModuleLoaded
- 3:0           - Module C:\WINDOWS\SYSTEM32\VCRUNTIME140D.dll at
address 0... - =
= [0x2] : Module Loaded at position: 4:0          - ModuleLoaded
- 4:0           - Module C:\WINDOWS\SYSTEM32\ucrtbased.dll at
address 0X118... - =
= [0x3] : Module Loaded at position: 5:0          - ModuleLoaded
- 5:0           - Module
C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -
=
= [0x4] : Module Loaded at position: 6:0          - ModuleLoaded
- 6:0           - Module
C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -
=
= [0x5] : Module Loaded at position: 7:0          - ModuleLoaded
- 7:0           - Module C:\WINDOWS\SYSTEM32\apphelp.dll at address
0X73770... - =
= [0x6] : Module Loaded at position: 8:0          - ModuleLoaded
- 8:0           - Module C:\WINDOWS\System32\KERNELBASE.dll at
address 0X73... - =
= [0x7] : Module Loaded at position: 9:0          - ModuleLoaded
- 9:0           - Module C:\WINDOWS\System32\KERNEL32.DLL at address
0X7590... - =
= [0x8] : Module Loaded at position: A:0          - ModuleLoaded
- A:0           - Module C:\WINDOWS\SYSTEM32\ntdll.dll at address
0X7707000... - =
= [0x9] : Thread created at D:0          - ThreadCreated
- D:0           -
- UID: 2, TID: 0x4C2C   =
= [0xa] : Thread terminated at 64:0          -
ThreadTerminated - 64:0           -
- UID: 2, TID: 0x4C2C   =
= [0xb] : Thread created at 69:0          - ThreadCreated
- 69:0           -
- UID: 3, TID: 0x4CFC   =
= [0xc] : Thread created at 6A:0          - ThreadCreated
- 6A:0           -
- UID: 4, TID: 0x27B0   =
= [0xd] : Thread terminated at 89:0          -
ThreadTerminated - 89:0           -
- UID: 4, TID: 0x27B0   =
= [0xe] : Thread terminated at 8A:0          -
ThreadTerminated - 8A:0           -
- UID: 3, TID: 0x4CFC   =
= [0xf] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0     - Module C:\Users\USER1\Documents\Visual Studio
```

```

2015\Proje... - =
= [0x10] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module
C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -
=
= [0x11] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\SYSTEM32\VCRUNTIME140D.dll at
address 0... - =
= [0x12] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module
C:\Users\USER1\AppData\Local\Dbg\UI\Fast.20170907... -
=
= [0x13] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\SYSTEM32\ucrtbased.dll at
address 0X118... - =
= [0x14] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\SYSTEM32\apphelp.dll at address
0X73770... - =
= [0x15] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\System32\KERNELBASE.dll at
address 0X73... - =
= [0x16] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\System32\KERNEL32.DLL at address
0X7590... - =
= [0x17] : Module Unloaded at position: FFFFFFFFFFFFFF:0 - ModuleUnloaded
- FFFFFFFFFFFFFF:0 - Module C:\WINDOWS\SYSTEM32\ntdll.dll at address
0X7707000... - =
=====
=====
```

Select any of the columns with a + sign to sort the output.

Use this LINQ query to display in grid format, the time position in the trace when threads were created (Type == "ThreadCreated").

```

dbgcmd

dx -g @$curprocess.TTD.Events.Where(t => t.Type == "ThreadCreated").Select(t
=> t.Thread)
=====
=====
= (+) UniqueId = (+) Id      = (+) Lifetime
= (+) ActiveTime =
=====
=====
= [0x0] : UID: 2, TID: 0x4C2C - 0x2          - 0x4c2c    - [0:0,
FFCCCCCC:0] - [D:0, 64:0]      =
= [0x1] : UID: 3, TID: 0x4CFC - 0x3          - 0x4cfcc   - [0:0, 8A:0]
- [69:0, 8A:0]      =
= [0x2] : UID: 4, TID: 0x27B0 - 0x4          - 0x27b0    - [0:0, 89:0]
- [6A:0, 89:0]      =
```

```
=====
=====
```

Use this LINQ query to display in grid format, the time positions in the trace when threads were terminated (Type == "ThreadTerminated").

```
dbgcmd

0:000> dx -g @$curprocess.TTD.Events.Where(t => t.Type ==
"ThreadTerminated").Select(t => t.Thread)
=====
=====
= (+) UniqueId = (+) Id      = (+) Lifetime
= (+) ActiveTime =
=====
= [0x0] : UID: 2, TID: 0x4C2C - 0x2          - 0x4c2c    - [0:0,
FFFFFFFFFFFE:0]      - [D:0, 64:0]      =
= [0x1] : UID: 4, TID: 0x27B0 - 0x4          - 0x27b0    - [0:0, 89:0]
- [6A:0, 89:0]      =
= [0x2] : UID: 3, TID: 0x4CFC - 0x3          - 0x4cfcc   - [0:0, 8A:0]
- [69:0, 8A:0]      =
=====
```

Sorting output to determine the longest running threads

Use this LINQ query to display in grid format, the approximate longest running threads in the trace.

```
dbgcmd

0:000> dx -g @$curprocess.TTD.Events.Where(e => e.Type ==
"ThreadTerminated").Select(e => new { Thread = e.Thread, ActiveTimeLength =
e.Thread.ActiveTime.MaxPosition.Sequence -
e.Thread.ActiveTime.MinPosition.Sequence }).OrderByDescending(t =>
t.ActiveTimeLength)
=====
= (+) Thread          = ActiveTimeLength =
=====
= [0x0]    - UID: 2, TID: 0x1750    - 0x364030    =
= [0x1]    - UID: 3, TID: 0x420C    - 0x360fd4    =
= [0x2]    - UID: 7, TID: 0x352C    - 0x35da46    =
= [0x3]    - UID: 9, TID: 0x39F4    - 0x34a5b5    =
= [0x4]    - UID: 11, TID: 0x4288   - 0x326199    =
= [0x5]    - UID: 13, TID: 0x21C8   - 0x2fa8d8    =
= [0x6]    - UID: 14, TID: 0x2188   - 0x2a03e3    =
= [0x7]    - UID: 15, TID: 0x40E8    - 0x29e7d0    =
= [0x8]    - UID: 16, TID: 0x124     - 0x299677    =
= [0x9]    - UID: 4, TID: 0x2D74    - 0x250f43    =
```

```
= [0xa] - UID: 5, TID: 0x2DC8 - 0x24f921 =
= [0xb] - UID: 6, TID: 0x3B1C - 0x24ec8e =
= [0xc] - UID: 10, TID: 0x3808 - 0xf916f =
= [0xd] - UID: 12, TID: 0x26B8 - 0x1ed3a =
= [0xe] - UID: 17, TID: 0x37D8 - 0xc65 =
= [0xf] - UID: 8, TID: 0x45F8 - 0x1a2 =
=====
```

Querying for read accesses to a memory range

Use the [TTD.Memory object](#) to query for to query for read accesses to a memory range.

The Thread Environment Block (TEB) is a structure that contains all the information regarding the state of a thread, including the result returned by GetLastError(). You can query this data structure by running `dx @$teb` for the current thread. One of TEB's members is a LastErrorValue variable, 4 bytes in size. We can reference the LastErrorValue member in the TEB using this syntax. `dx &@$teb->LastErrorValue`.

The example query shows how to find every read operation done in that range in memory, select all the reads that happen before the a dialog was created and then sort the result to find the last read operation.

```
dbgcmd

0:000> dx @$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb-
>LastErrorValue + 0x4, "r")
@$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb->LastErrorValue +
0x4, "r")
[0x0]
[0x1]
[0x2]
[0x3]
```

If in our trace a "dialog" event has taken place we can run a query to find every read operation done in that range in memory, select all the reads that happen before the dialog got created and then sort the result to find the last read operation. Then time travel to that point in time by calling SeekTo() on the resulting time position.

```
dbgcmd

:000> dx @$cursession.TTD.Memory(&@$teb->LastErrorValue, &@$teb-
>LastErrorValue + 0x4, "r").Where(m => m.TimeStart < @$dialog).OrderBy(m =>
m.TimeStart).Last().TimeEnd.SeekTo()
Setting position: 458300:37
ModLoad: 6cee0000 6cf5b000 C:\WINDOWS\system32\uxtheme.dll
ModLoad: 75250000 752e6000 C:\WINDOWS\System32\OLEAUT32.dll
```

```
ModLoad: 76320000 7645d000 C:\WINDOWS\System32\MSCTF.dll  
ModLoad: 76cc0000 76cce000 C:\WINDOWS\System32\MSASN1.dll
```

GitHub TTD Query Lab

For a tutorial on how to debug C++ code using a Time Travel Debugging recording using queries to find information about the execution of the problematic code in question, see [WinDbg-Samples - Time Travel Debugging and Queries ↴](#).

All of the code used in the lab is available here: [https://github.com/Microsoft/WinDbg-Samples/tree/master/TTDQueries/app-sample ↴](https://github.com/Microsoft/WinDbg-Samples/tree/master/TTDQueries/app-sample).

Troubleshooting TTD Queries

"UnknownOrMissingSymbols" as the function names

The data model extension needs full symbol information in order to provide function names, parameter values, etc. When full symbol information is not available the debugger uses "UnknownOrMissingSymbols" as the function name.

- If you have private symbols you will get the function name and the correct list of parameters.
- If you have public symbols you will get the function name and a default set of parameters - four unsigned 64-bit ints.
- If you have no symbol information for the module you are querying, then "UnknownOrMissingSymbols" is used as the name.

TTD Queries for calls

There can be several reasons that a query doesn't return anything for calls to a DLL.

- The syntax for the call isn't quite right. Try verifying the call syntax by using the [x \(Examine Symbols\) ↴ <call>](#). If the module name returned by x is in uppercase, use that.
- The DLL is not loaded yet and is loaded later in the trace. To work around this travel to a point in time after the DLL is loaded and redo the query.
- The call is inlined which the query engine is unable to track.
- The query pattern uses wildcards which returns too many functions. Try to make the query pattern more specific so that the number of matched functions is small enough.

See Also

[Using LINQ With the debugger objects](#)

[dx \(Display Debugger Object Model Expression\)](#)

[Time Travel Debugging - Overview](#)

[Time Travel Debugging - JavaScript Automation](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Calls Objects

Article • 11/21/2024

Description

TTD Calls objects are used to give information about function calls that occur over the course of a trace.

Parameters

[+] Expand table

Property	Description
Function!SymbolName	One or more contained in double quotes, separated by a comma. For example dx @\$cursession.TTD.Calls("module!symbol1", "module!symbol2", ...)

Properties

[+] Expand table

Property	Description
EventType	The type of event. This is "Call" for all TTD Calls objects.
ThreadId	The OS thread ID of thread that made the request.
UniqueThreadId	A unique ID for the thread across the trace. Regular thread IDs can get reused over the lifetime of a process but UniqueThreadIds cannot.
Function	The symbolic name of the function.
FunctionAddress	The function's address in memory.
ReturnValue	The return value of the function. If the function has a void type, this property will not be present.

Children

[+] Expand table

Object	Description
Parameters[]	An array containing the parameters passed to the function. The number of elements varies based on the type signature of the function.
TimeStart	A position object that describes the position at the start of the call.
TimeEnd	A position object that describes the position at the end of the call.

Remarks

Time travel debugging uses symbol information provided in the PDBs to determine the number of parameters for a function and their types, the return value type, and the calling convention. In the event that symbol information is not available or the symbols have been restricted to public symbol information, it is still possible to do queries. The time travel query engine will make some assumptions in this scenario:

- There are four 64-bit unsigned integer parameters to the function
- The return value is a 64-bit unsigned integer
- The function name is set to a fixed string: "UnknownOrMissingSymbols"

These assumptions allow queries to be made in the absence of adequate symbol information. However, for best results use full PDB symbols when possible.

Note that the Calls function does computation, and depending on the size of the trace, it can take a while to run. CPU usage will spike during the computation and watching CPU usage in task manager, gives an indication that the computation is progressing. The query results are cached in memory so subsequent queries against previously queried calls are significantly faster.

Example Usage

This example shows the calls object for ucrtbase!initterm.

```
dbgcmd

0:000> dx -r2 @$cursession.TTD.Calls("ucrtbase!initterm")
@$cursession.TTD.Calls("ucrtbase!initterm")
[0x0]
    EventType      : Call
    ThreadId       : 0x2074
    UniqueThreadId : 0x2
    TimeStart      : 1E:5D0
    TimeEnd        : 2D:E
    Function       : ucrtbase!_initterm
```

```
FunctionAddress : 0x7ffb345825d0
ReturnAddress   : 0x7ff6a521677e
Parameters
SystemTimeStart : Friday, January 12, 2024 21:18:40.799
SystemTimeEnd   : Friday, January 12, 2024 21:18:44.26
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?



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

TTD Heap Objects

Article • 11/21/2024

Description

TTD Heap objects are used to give information about heap calls that occur over the course of a trace.

Properties

Every heap object will have these properties.

[+] Expand table

Property	Description
Action	Describes the action that occurred. Possible values are: Alloc, ReAlloc, Free, Create, Protect, Lock, Unlock, Destroy.
Heap	The handle for the Win32 heap.

Conditional properties

Depending on the heap object, it may have some of the properties below.

[+] Expand table

Property	Description
Address	The address of the allocated object.
PreviousAddress	The address of the allocated object before it was reallocated. If Address is not the same as PreviousAddress then the reallocation caused the memory to move.
Size	The size and/or requested size of an allocated object.
BaseAddress	The address of an allocated object in the heap. It can represent the address which will be freed (Free) or the address of the object before it is reallocated (ReAlloc.)
Flags	Meaning depends on the API.

Property	Description
Result	The result of the heap API call. Non-zero means success and zero means failure.
ReserveSize	Amount of memory to reserve for the heap.
CommitSize	Initial committed size for the heap.
MakeReadOnly	A non-zero value indicates a request to make the heap read-only; A zero value indicates the heap should be read-write.

Children

[\[+\] Expand table](#)

Object	Description
TimeStart	A position object that describes the position at the start of the allocation.
TimeEnd	A position object that describes the position at the end of the allocation.

Example Usage

Use this [dx \(Display Debugger Object Model Expression\)](#) command to display the heap memory in a grid using the -g option.

```
dbgcmd

0:0:000> dx -g @$cursession.TTD.Data.Heap()
=====
=====
=           = Action      = Heap          = Address      =
Size       = Flags     = (+) TimeStart = (+) TimeEnd = Result = PreviousAddress
=
=====
=====
= [0x0] : [object Object] - Alloc      - 0xaf0000      - 0xb0cf0d0      -
0x4c      - 0x0       - FAB:17B1      - FAD:40      -      -
=
= [0x1] : [object Object] - Alloc      - 0xaf0000      - 0xb07210      -
0x34      - 0x8       - FB1:9       - FB3:74      -      -
=
= [0x2] : [object Object] - Alloc      - 0xaf0000      - 0xb256d8      -
0x3c      - 0x8       - E525:174     - E526:E1      -      -
=
```

The output can be described as “normalized data” because there is a chosen set of APIs that represent heap operations. The data that is extracted from the appropriate parameters, is presented in a uniform manner.

Clicking on TimeStart or TimeEnd will navigate you to that point in the trace.

Click on the parameters field next to a specific entry, to display available parameter information.

```
dbgcmd

dx -r1 @$cursession.TTD.Data.Heap()[2].@"Parameters"
@$cursession.TTD.Data.Heap()[2].@"Parameters"
[0x0] : 0x16c7d780000
[0x1] : 0x280000
[0x2] : 0x20
[0x3] : 0x0
...
```

Increase the recursion -r value to display additional information.

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Memory Objects

Article • 11/14/2024

Description

TTD Memory is a method that takes beginAddress, endAddress and dataAccessMask parameters and returns a collection of memory objects that contain memory access information.

Parameters

[+] Expand table

Property	Description
beginAddress	The beginning address of the memory object prefaced with 0x.
endAddress	The ending address of the memory object prefaced with 0x.
dataAccessMask	The data access mask contained in double quotes. This can be r for read, w for write, e for execute and c for change.

Children

[+] Expand table

Object	Description
EventType	The type of event. This is "MemoryAccess" for all TTD.Memory objects.
ThreadId	The OS thread ID of thread that made the request.
UniqueThreadId	A unique ID for the thread across the trace. Regular thread IDs can get reused over the lifetime of a process but UniqueThreadIds cannot.
TimeStart	A position object that describes the position when memory access was made.
TimeEnd	A position object that describes the position when memory access was made. This will always be the same as the TimeStart for TTD.Memory objects.
AccessType	The access type - Read, Write or Execute.
IP	The instruction pointer of the code that made the memory access.

Object	Description
Address	The Address that was read / written to / executed and will be in the range of [beginAddress, endAddress) from the parameters to .Memory(). Note that the interval is half-open. That is, none of the returned events will have an address matching endAddress but there could be events matching endAddress – 1.
Size	The size of the read/write/execute in bytes. This will typically be 8 bytes or less. In the event of code execution, it is the number of bytes in the instruction that was executed.
Value	The value that was read, written or executed. In the case of execution, it contains the code bytes for the instruction. Note the instruction bytes are listed in MSB order by the disassembler but will be stored in value in LSB order.

Remarks

The following access types are allowed in TTD.Memory queries:

- r - read
- w - write
- rw - read / write
- e - execute
- rwe - read / write / execute
- ec - execute /change

Note that this is a function that does computation, so it takes a while to run.

Example Usage

This example shows a grid display of all the positions in the trace where the four bytes of memory starting at 0x00a4fca0 were read access occurred. Click on any entry to drill down on each occurrence of memory access.

```
dbgcmd
```

```
dx -g @$cursession.TTD.Memory(0x00a4fca0,0x00a4fca4, "r")
```

	(±) EventType	(±) ThreadId	(±) UniqueThreadId	(±) TimeStart	(±) TimeEnd	(±) AccessType	(±) IP	(±) Address	(±) Size	(±) Value
[0x0]	MemoryAccess	0x3168	0x2	15:16	15:16	Read	0x77a02196	0xa4fc0	0x4	0xa4fcf8
[0x1]	MemoryAccess	0x3168	0x2	15:18	15:18	Read	0x77a02198	0xa4fc0	0x4	0x77a26768
[0x2]	MemoryAccess	0x3168	0x2	15:36	15:36	Read	0x779e4385	0xa4fc0	0x4	0x779e0c21
[0x3]	MemoryAccess	0x3168	0x2	1E:26	1E:26	Read	0x779e0d07	0xa4fc0	0x4	0xa4fcf8
[0x4]	MemoryAccess	0x3168	0x2	2B:183	2B:183	Read	0x690b9c21	0xa4fc0	0x4	0x0
[0x5]	MemoryAccess	0x3168	0x2	3F:6	3F:6	Read	0x7562500a	0xa4fc0	0x4	0x1f000
[0x6]	MemoryAccess	0x3168	0x2	43:3	43:3	Read	0x79df9a9b	0xa4fc0	0x4	0x117a9480
[0x7]	MemoryAccess	0x3168	0x2	45:4	45:4	Read	0x779bc75b	0xa4fc0	0x4	0x799123f0
[0x8]	MemoryAccess	0x3168	0x2	47:3	47:3	Read	0x779df73b	0xa4fc0	0x4	0x117a9480
[0x9]	MemoryAccess	0x3168	0x2	47:A	47:A	Read	0x779df74b	0xa4fc0	0x4	0x117a9480
[0xa]	MemoryAccess	0x3168	0x2	4B:2F	4B:2F	Read	0xb0a688	0xa4fc0	0x4	0xa4fd00
[0xb]	MemoryAccess	0x3168	0x2	4B:118	4B:118	Read	0xb02e96	0xa4fc0	0x4	0x0
[0xc]	MemoryAccess	0x3168	0x2	4B:11C	4B:11C	Read	0xb02ea1	0xa4fc0	0x4	0x0
[0xd]	MemoryAccess	0x3168	0x2	4B:928	4B:928	Read	0x690c2bec	0xa4fc0	0x4	0x69191a60
[0xe]	MemoryAccess	0x3168	0x2	4B:939	4B:939	Read	0x690ca901	0xa4fc0	0x4	0x6918d19
[0xf]	MemoryAccess	0x3168	0x2	4B:941	4B:941	Read	0x779c3418	0xa4fc0	0x4	0x69191a80
[0x10]	MemoryAccess	0x3168	0x2	5D:113	5D:113	Read	0x690795e5	0xa4fc0	0x4	0x0
[0x11]	MemoryAccess	0x3168	0x2	5F:18	5F:18	Read	0x690795f0	0xa4fc0	0x4	0x0
[0x12]	MemoryAccess	0x3168	0x2	64:60	64:60	Read	0x690ca8fc	0xa4fc0	0x4	0x69191a60
[0x13]	MemoryAccess	0x3168	0x2	7A:CC	7A:CC	Read	0x77a340d9	0xa4fc0	0x4	0xf0
[0x14]	MemoryAccess	0x3168	0x2	B9:CC	B9:CC	Read	0x77a340d9	0xa4fc0	0x4	0x0

You can click on the TimeStart fields in any of the events in the grid display, to display information for that event.

```
dbgcmd

0:000> dx -$cursession.TTD.Memory(0x00a4fc0,0x00a4fc4, "r")
[16].TimeStart
@$cursession.TTD.Memory(0x00a4fc0,0x00a4fc4, "r")[16].TimeStart
: 5D:113 [Time Travel]
Sequence : 0x5d
Steps : 0x113
```

To move to the position in the trace that the event occurred, click on [Time Travel].

```
dbgcmd

0:000> dx @$cursession.TTD.Memory(0x00a4fc0,0x00a4fc4, "r")
[16].TimeStart.SeekTo()
@$cursession.TTD.Memory(0x00a4fc0,0x00a4fc4, "r")[16].TimeStart.SeekTo()
(27b8.3168): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 5D:113

eax=0000004c ebx=00dd0000 ecx=00a4f89c edx=00a4f85c esi=00a4f89c
edi=00b61046
eip=690795e5 esp=00a4f808 ebp=00a4f818 iopl=0          nv up ei pl nz na pe
nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
efl=00000206
690795e5 ffb604040000      push    dword ptr [esi+404h]
ds:002b:00a4fc0=00000000
```

In this example, all of the positions in the trace where the four bytes of memory starting at 0x1bf7d0 were read/write accessed are listed. Click on any entry to drill down on each occurrence of memory access.

```
dbgcmd
```

```
0:000> dx @$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
@$cursession.TTD.Memory(0x1bf7d0,0x1bf7d4, "rw")
[0x0]
[0x1]
[0x2]
[0x3]
...
```

In this example all of the positions in the trace where the four bytes of memory starting at 0x13a1710 were execute/change accessed are listed. Click on any occurrence to drill down on for additional information on each occurrence of memory access.

dbgcmd

```
0:000> dx -r1 @$cursession.TTD.Memory(0x13a1710,0x13a1714, "ec")[0]
@$cursession.TTD.Memory(0x13a1710,0x13a1714, "ec")[0]
  EventType      : MemoryAccess
  ThreadId       : 0x1278
  UniqueThreadId : 0x2
  TimeStart      : 5B:4D [Time Travel]
  TimeEnd        : 5B:4D [Time Travel]
  AccessType     : Execute
  IP             : 0x13a1710
  Address        : 0x13a1710
  Size           : 0x1
  Value          : 0x55
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Event Objects

Article • 11/21/2024

Description

TTD Event objects are used to give information about important events that happened during a time travel trace.

Properties

[Expand table

Property	Description
Type	Describes the type of event that happened. Possible values are: ThreadCreated, ThreadTerminated, ModuleLoaded, ModuleUnloaded, Exception

Children

[Expand table

Object	Description
Position	A position object that describes the position the event occurred.
Module*	A module object containing information about the module that was loaded or unloaded.
Thread*	A thread object containing information about the thread that was created or terminated.
Exception*	An exception object containing information about the exception that was hit.

* - Existence of these child objects depends on the type of event

Example Usage

dbgcmd

```
0:000> dx -r2 @$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e => e.Exception)
```

```
@$curprocess.TTD.Events.Where(t => t.Type == "Exception").Select(e =>
e.Exception)
[0x0]          : Exception of type CPlusPlus at PC: 0X777663B0
    Position      : 13B7:0 [Time Travel]
    Type          : CPlusPlus
    ProgramCounter : 0x777663b0
    Code           : 0xe06d7363
    Flags          : 0x1
    RecordAddress  : 0x0
[0x1]          : Exception of type Hardware at PC: 0XF1260D0
    Position      : BC0F:0 [Time Travel]
    Type          : Hardware
    ProgramCounter : 0xf1260d0
    Code           : 0x80000003
    Flags          : 0x0
    RecordAddress  : 0x0
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?



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

TTD Exception Objects

Article • 11/21/2024

Description

TTD Exception objects are used to provide information about event exceptions that happened during a trace session.

Properties

[+] Expand table

Property	Description
Type	Describes the type of exception. Possible values are "Software" and "Hardware".
ProgramCounter	The instruction where the exception was thrown.
Code	The code of the exception.
Flags	The exception flags.
RecordAddress	Where in memory you can find the record of the exception.

Children

[+] Expand table

Object	Description
Position	A position object that describes the position the exception occurred.

Example Usage

```
dbgcmd
```

```
0:003> dx -r1 @$curprocess.TTD.Events.Where(t => t.Type == "Exception")  
[0].Exception  
@$curprocess.TTD.Events.Where(t => t.Type == "Exception")[0].Exception  
: Exception 0x80010012 of type Software at PC: 0X7FF9F6DC8670  
    Position      : 36A7:0 [Time Travel]
```

```
Type           : Software
ProgramCounter : 0x7ff9f6dc8670
Code           : 0x80010012
Flags          : 0x1
RecordAddress  : 0x0
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?



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

TTD Position Objects

Article • 11/21/2024

Description

Position objects are used to describe a position in a time travel trace. A position object is normally described by two hexadecimal numbers separated by a colon. The first of the hexadecimal numbers is the *Sequence* and the second is the *Steps*.

A position of FFFFFFFFFFFFFF:0 indicates the end of the trace.

Properties

[+] Expand table

Property	Description
Sequence	The sequencing point relevant to the position.
Steps	The number of steps from the sequence point in this thread to get to this position.

Methods

[+] Expand table

Method	Description
SeekTo()	Time travels to this position in the trace.

Example Usage

dbgcmd

```
0:003> dx -r1 @$create("Debugger.Models.TTD.Position", 14006, 0)
@$create("Debugger.Models.TTD.Position", 14006, 0) : 36B6:0
[Time Travel]
    Sequence      : 0x36b6
    Steps         : 0x0
    SeekTo        : [Method which seeks to time position]
```

ToSystemTime [Method which obtains the approximate system time at a given position]

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Range Objects

Article • 11/21/2024

Description

TTD Range objects are used to give information about a range of time in a trace. These are generally used to describe the lifetime of a [TTD thread object](#) during a TTD session.

Children

[\[\] Expand table](#)

Object	Description
MinPosition	A position object that describes the earliest position relevant to the range.
MaxPosition	A position object that describes the latest position relevant to the range.

Example Usage

In this example the MinPosition and MaxPosition objects are shown for Lifetime and ActiveTime associated with a thread.

```
dbgcmd

0:003> dx -r1 @$curprocess.TTD.Threads[5]
@$curprocess.TTD.Threads[5] : UID: 7, TID: 0x2580
    UniqueId      : 0x7
    Id            : 0x2580
    Lifetime      : [BAF:0, FFFFFFFFFFFFFE:0]
    ActiveTime    : [BB2:0, C6A:0]
    GatherMemoryUse [Gather inputs, outputs and memory used by a range of
execution within a thread]

0:003> dx -r1 @$curprocess.TTD.Threads[5].Lifetime
@$curprocess.TTD.Threads[5].Lifetime : [BAF:0,
FFFFFFFFFFFFFFE:0]
    MinPosition   : BAF:0 [Time Travel]
    MaxPosition   : FFFFFFFFFFFFFE:0 [Time Travel]

0:003> dx -r1 @$curprocess.TTD.Threads[5].ActiveTime
@$curprocess.TTD.Threads[5].ActiveTime : [BB2:0, C6A:0]
```

```
MinPosition      : BB2:0 [Time Travel]
MaxPosition      : C6A:0 [Time Travel]
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Thread Objects

Article • 11/21/2024

Description

TTD Thread objects are used to give information about threads and their lifetime during a time travel trace.

Properties

[+] Expand table

Property	Description
Uniqueld	A unique ID for the thread across the trace.
Id	The TID of the thread.

Children

[+] Expand table

Object	Description
LifeTime	A TTD range object that describes the lifetime of the thread.
ActiveTime	A TTD range object that describes the time the thread was active.

Example Usage

Use this dx command to display all of the threads in the array.

```
dbgcmd

0:0:000> dx -g @$curprocess.TTD.Threads
=====
=====
=          = (+) UniqueId = (+) Id      = (+) Lifetime
= (+) ActiveTime      =
=====
=====
= [0x0] : UID: 2, TID: 0x2428 - 0x2           - 0x2428      - [0:0, 6F0C4:0]
```

```

- [50C63:0, 6F0C4:0]      =
= [0x1] : UID: 3, TID: 0x3520 - 0x3          - 0x3520    - [0:0,
FFFFFFFFFFFFFFFE:0]      - [5115A:0, 56B07:0]      = 
= [0x2] : UID: 4, TID: 0x18E8 - 0x4          - 0x18e8    - [0:0,
FFFFFFFFFFFFFFFE:0]      - [52F65:0, 56B1E:0]      = 
= [0x3] : UID: 5, TID: 0x5690 - 0x5          - 0x5690    - [0:0,
FFFFFFFFFFFFFFFE:0]      - [5300D:0, 5D4FA:0]      = 
= [0x4] : UID: 6, TID: 0x46FC - 0x6          - 0x46fc    - [0:0,
FFFFFFFFFFFFFFFE:0]      - [53782:0, 5433B:0]      = 
= [0x5] : UID: 7, TID: 0x58D0 - 0x7          - 0x58d0    - [0:0,
FFFFFFFFFFFFFFFE:0]      - [542FE:0, 543B9:0]      = 
= [0x6] : UID: 8, TID: 0x950 - 0x8           - 0x950     - [0:0,
FFFFFFFFFFFFFFFE:0]      - [543C4:0, 544B8:0]      = 
= [0x7] : UID: 9, TID: 0x4514 - 0x9           - 0x4514    - [0:0, 6D61B:0]
- [5DBBD:0, 6D61B:0]      = 
=====
=====
```

Use this dx command to display information about the first thread in the array.

```
dbgcmd

0:0:000 dx -r2 @$curprocess.TTD.Threads[0]
@$curprocess.TTD.Threads[0] : UID: 2, TID: 0x2428
  UniqueId      : 0x2
  Id            : 0x2428
  Lifetime      : [0:0, 6F0C4:0]
    MinPosition   : Min Position [Time Travel]
    MaxPosition   : 6F0C4:0 [Time Travel]
  ActiveTime     : [50C63:0, 6F0C4:0]
    MinPosition   : 50C63:0 [Time Travel]
    MaxPosition   : 6F0C4:0 [Time Travel]
```

The [Time Travel] links provide a link to SeekTo() the specific position in the trace when the thread was active.

```
dbgcmd

0:0:000> dx
@$curprocess.TTD.Threads[0].@"ActiveTime".@"MinPosition".SeekTo()
Setting position: 50C63:0
@$curprocess.TTD.Threads[0].@"ActiveTime".@"MinPosition".SeekTo()
(40b4.2428): Break instruction exception - code 80000003 (first/second
chance not available)
Time Travel Position: 50C63:0
ntdll!NtTestAlert+0x14:
00007ffb`e3e289d4 c3          ret
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Module Objects

Article • 11/21/2024

Description

TTD Module objects are used to give information about modules that were loaded and unloaded during a trace session.

Properties

[\[\] Expand table](#)

Property	Description
Name	The name and path of the module.
Address	The address where the module was loaded.
Size	The size of the module in bytes.
Checksum	The checksum of the module.
Timestamp	The timestamp of the module.

Example Usage

```
dbgcmd

0:005> dx -r1 @$cursession.Processes[24848].Modules[10]
@$cursession.Processes[24848].Modules[10] : C:\Program
Files\WindowsApps\Microsoft.VCLibs.140.00_14.0.30704.0_x64_8wekyb3d8bbwe\VC
RUNTIME140_APP.dll [Force Symbol Reload]
    BaseAddress      : 0x7ff9adae0000
    Name            : C:\Program
Files\WindowsApps\Microsoft.VCLibs.140.00_14.0.30704.0_x64_8wekyb3d8bbwe\VC
RUNTIME140_APP.dll
    Size            : 0x1b000
    SymbolType      : Deferred
    Attributes
    Contents
    Symbols         : [SymbolModule]VCRUNTIME140_APP
```

See Also

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

[dx \(Display Debugger Object Model Expression\)](#)

Feedback

Was this page helpful?

 Yes

 No

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

TTD Collection Objects

Article • 11/21/2024

Description

The collection model objects allow for common manipulation of debugger model objects that support the collection objects.

Children

[+] Expand table

Object	Description
MinPosition	A position object that describes the earliest position relevant to the range.

TTD Collection Object Methods

Contains(OtherString) -Method which returns whether the string contains a given substring.

EndsWith(OtherString) -Method which returns whether the string ends with a given string.

IndexOf(OtherString) -Method which returns the index of the first occurrence of a substring in the given string. If no such occurrence exists, -1 is returned.

LastIndexOf(OtherString) -Method which returns the index of the last occurrence of a substring in the given string. If no such occurrence exists, -1 is returned.

Length - Property which returns the length of the string.

PadLeft(TotalWidth) - Method which right aligns the string to the specified width by inserting spaces at the left of the string.

PadRight(TotalWidth) - Method which left aligns the string to the specified width by inserting spaces at the right of the string.

Remove(StartPos, [Length]) - Method which removes all characters beginning at the specified position from the string. If an optional length is supplied, only that many characters after the starting position are removed.

Replace(SearchString, ReplaceString) - Method which replaces every occurrence of a specified search string with a replacement string.

StartsWith(OtherString) - Method which returns whether the string starts with a given string.

Substring(StartPos, [Length]) - Method which retrieves a substring from the given string. The substring starts at a specified character position and continues to the end of the string or for the optionally specified length.

ToLower() - Returns a copy of this string converted to lowercase.

ToUpper() - Returns a copy of this string converted to uppercase.

Example Usage

This LINQ query displays the load event(s) of a particular module, that contains, ntdll.dll.

```
dbgcmd

0:000> dx @$curprocess.TTD.Events.Where(t => t.Type ==
"ModuleUnloaded").Where(t => t.Module.Name.Contains("ntdll.dll"))
```

See Also

[Using LINQ With the debugger objects](#)

[dx \(Display Debugger Object Model Expression\)](#)

[Time Travel Debugging - Introduction to Time Travel Debugging objects](#)

[Time Travel Debugging - Overview](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time Travel Debugging - JavaScript Automation

Article • 11/19/2024



You can use JavaScript automation to work with TTD traces in a number of ways, such as command automation or using queries to locate event data from the trace file.

For general information about working with JavaScript see [JavaScript Debugger Scripting](#). There are also [JavaScript Debugger Example Scripts](#).

JavaScript TTD Command Automation

One way to use JavaScript for TTD automation, is to send commands to automate working with time travel trace files.

Moving in a trace file

This JavaScript shows how to move to the start of a time travel trace using the `!tt` command.

```
JavaScript

var dbgControl = host.namespace.Debugger.Utility.Control;
dbgControl.ExecuteCommand("!tt 0",false);
host.diagnostics.debugLog("">>>> Sent command to move to the start of the TTD
file \n");
```

We can make this into a `ResetTrace` function, and save it as `MyTraceUtils.js`, using the JavaScript UI in WinDbg.

```
JavaScript

// My Trace Utils
// WinDbg TTD JavaScript MyTraceUtilsCmd Sample

"use strict";

function MyTraceUtilsCmd()
```

```
{  
    var dbgControl = host.namespace.Debugger.Utility.Control;  
    dbgControl.ExecuteCommand("!tt 0", false);  
    host.diagnostics.debugLog(">>> Sent command to move to the start of the  
    TTD file \n");  
}
```

After a TTD file is loaded in WinDbg, call the function ResetTraceCmd() function using the dx command in the debugger command window.

```
dbgcmd  
  
0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ResetTraceCmd()  
>>> Sent command to move to the start of the TTD file  
Debugger.State.Scripts.MyTraceUtils.Contents.ResetTraceCmd()
```

Limitations of sending commands

But for all but the simplest situations, the approach of sending commands has drawbacks. It relies on the use of text output. And parsing that output leads to code that is brittle and hard to maintain. A better approach is to use the TTD objects directly.

The following example shows how to use the objects directly to complete the same task using the objects directly.

```
JavaScript  
  
// My Trace Utils  
// WinDbg TTD JavaScript ResetTrace Sample  
  
"use strict";  
  
function ResetTrace()  
{  
    host.currentProcess.TTD.SetPosition(0);  
    host.diagnostics.debugLog(">>> Set position to the start of the TTD file  
\n");  
}
```

Running this code shows that we are able to move to the start of the trace file.

```
dbgcmd  
  
0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ResetTrace()  
(948.148c): Break instruction exception - code 80000003 (first/second chance  
not available)
```

```
Time Travel Position: F:0
>>> Set position to the start of the TTD file
```

In this example ResetTraceEnd function, the position is set to the end of the trace and the current and new position is displayed using the `currentThread.TTD` Position object.

JavaScript

```
// WinDbg TTD JavaScript Sample to Reset Trace using objects directly
// and display current and new position

function ResetTraceEnd()
{
    var PositionOutputStart = host.currentThread.TTD.Position;
    host.diagnostics.debugLog(">>> Current position in trace file: " +
PositionOutputStart +"\n");
    host.currentProcess.TTD.SetPosition(100);
    var PositionOutputNew = host.currentThread.TTD.Position;
    host.diagnostics.debugLog(">>> New position in trace file: " +
PositionOutputNew +"\n");
}
```

Running this code displays the current and new position.

dbgcmd

```
0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ResetTraceEnd()
>>> Current position in trace file: F:0
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: D3:1
>>> New position in trace file: D3:1
```

In this expanded sample, the starting and ending position values are compared to see if the position in the trace changed.

JavaScript

```
// WinDbg TTD JavaScript ResetTraceEx Sample

"use strict";

function ResetTraceEx()
{
    const PositionOutputStart = host.currentThread.TTD.Position;
    host.diagnostics.debugLog(">>> Current position in trace file: " +
PositionOutputStart +"\n");
```

```

host.currentProcess.TTD.SetPosition(0);

const PositionOutputNew = host.currentThread.TTD.Position;
host.diagnostics.debugLog(">>> New position in trace file: " +
PositionOutputNew +"\n");

if (parseInt(PositionOutputStart,16) != parseInt(PositionOutputNew,16))
{
    host.diagnostics.debugLog(">>> Set position to the start of the TTD
file \n");
}
else
{
    host.diagnostics.debugLog(">>> Position was already set to the start
of the TTD file \n");
}
}

```

In this example run, a message is displayed that we were all ready at the start of the trace file.

```

dbgcmd

0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ResetTraceEx()
>>> Current position in trace file: F:0
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: F:0
>>> New position in trace file: F:0
>>> Position was already set to the start of the TTD file

```

To test the script use the `!tt` command to navigate half way in the trace file.

```

dbgcmd

0:000> !tt 50
Setting position to 50% into the trace
Setting position: 71:0

```

Running the script now displays the proper message that indicates that the position was set to the start of the TTD trace.

```

dbgcmd

0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ResetTraceEx()
>>> Current position in trace file: 71:0
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)

```

```
Time Travel Position: F:0
>>> New position in trace file: F:0
>>> Set position to the start of the TTD file
```

Indexing a time travel trace file

If just a trace file is copied over to a different PC, it will need to be re-indexed. For more information, see [Time Travel Debugging - Working with Trace Files](#).

This code shows an example IndexTrace function that displays how long it takes to re-index a trace file.

JavaScript

```
function IndexTrace()
{
    var timeS = (new Date()).getTime();
    var output = host.currentProcess.TTD.Index.ForceBuildIndex();
    var timeE = (new Date()).getTime();
    host.diagnostics.debugLog("\n>>> Trace was indexed in " + (timeE - timeS) + " ms\n");
}
```

Here is the output from a small trace file.

dbgcmd

```
0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.IndexTrace()

>>> Trace was indexed in 2 ms
```

Adding a try catch statement

To check to see if errors were raised when the indexing was run, enclose the indexing code in a try catch statement.

JavaScript

```
function IndexTraceTry()
{
    var timeS = (new Date()).getTime();
    try
    {
        var IndexOutput = host.currentProcess.TTD.Index.ForceBuildIndex();
        host.diagnostics.debugLog("\n>>> Index Return Value: " +
```

```

IndexOutput + "\n");
    var timeE = (new Date()).getTime();
    host.diagnostics.debugLog("\n>>> Trace was successfully indexed in
" + (timeE - timeS) + " ms\n");
}

catch(err)
{
    host.diagnostics.debugLog("\n>>> Index Failed! \n");
    host.diagnostics.debugLog("\n>>> Index Return Value: " +
IndexOutput + "\n");
    host.diagnostics.debugLog("\n>>> Returned error: " + err.name +
"\n");
}
}

```

Here is the script output if the indexing is successful.

```

dbgcmd

0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.IndexTraceTry()

>>> Index Return Value: Loaded

>>> Trace was successfully indexed in 1 ms

```

If the trace can't be indexed, for example if the trace is not loaded in the debugger, the catch loop code is run.

```

dbgcmd

0:007> dx Debugger.State.Scripts.MyTraceUtils.Contents.IndexTraceTry()

>>> Index Failed!

>>> Index Return Value: undefined

>>> Returned error: TypeError

```

JavaScript TTD Objects Queries

A more advanced use of JavaScript and TTD is to query the time travel objects to locate specific calls or events that have occurred in the trace. For more information about the TTD objects see:

[Introduction to Time Travel Debugging objects](#)

Native Debugger Objects in JavaScript Extensions - Debugger Object Details

The `dx` command displays information from the debugger data model and supports queries using LINQ syntax. Dx is very useful to query the objects in realtime. This allows for the prototyping of the desired query that can be then automated using JavaScript. The `dx` command provides tab completion, which can be helpful when exploring the object model. For general information on working with LINQ queries and debugger objects, see [Using LINQ With the debugger objects](#).

This `dx` command, counts all the calls to a certain API, in this example `GetLastError`.

```
dbgcmd

0:000> dx @$cursession.TTD.Calls("kernelbase!GetLastError").Count()

@$cursession.TTD.Calls("kernelbase! GetLastError").Count() : 0x12
```

This command looks in the entire time travel trace to see when `GetLastError` was called.

```
dbgcmd

0:000> dx @$cursession.TTD.Calls("kernelbase!GetLastError").Where(c =>
c.ReturnValue != 0)

@$cursession.TTD.Calls("kernelbase!GetLastError").Where(c => c.ReturnValue
!= 0)
[0x0]
[0x1]
[0x2]
[0x3]
```

String comparisons for TTD.Calls Object to locate calls

This example command, shows how to use string comparisons to locate specific calls. In this example, the query looks for the string "OLE" in the `lpFileName` parameter of the [CreateFileW function](#).

```
dbgcmd

dx -r2 @$cursession.TTD.Calls("kernelbase!CreateFileW").Where(x =>
x.Parameters.lpFileName.ToString("su").Contains("OLE"))
```

Add a `.Select` statement to print `Timestart` and the value of the `lpFileName` parameter.

```
dbgcmd
```

```
dx -r2 @$cursession.TTD.Calls("kernelbase!CreateFileW").Where(x =>
x.Parameters.lpFileName.ToString("su").Contains("OLE")).Select(x =>
new { TimeStart = x.TimeStart, lpFileName = x.Parameters.lpFileName })
```

This generates this output, if a [TTD.Calls](#) object is found that contains the target information.

```
dbgcmd

[0x0]
    TimeStart      : 6E37:590
    lpFileName     : 0x346a78be90 : "C:\WINDOWS\SYSTEM32\OLEACCRC.DLL"
[Type: wchar_t *]
```

Displaying the number of calls in a trace

After you have used the dx command to explore objects you want to work with, you can automate their use with JavaScript. In this simple example, the [TTD.Calls](#) object is used to count calls to *kernelbase!GetLastError*.

```
JavaScript

function CountLastErrorCalls()
{
    var LastErrorCalls =
host.currentSession.TTD.Calls("kernelbase!GetLastError");
    host.diagnostics.debugLog(">>> GetLastError calls in this TTD recording:
" + LastErrorCalls.Count() +"\n");
}
```

Save the script in a TTDUtils.js file and call it using the dx command to display a count of the number of kernelbase!GetLastError in the trace file.

```
dbgcmd

0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.CountLastErrorCalls()
>>> GetLastError calls in this TTD recording: 18
```

Displaying the frames in a stack

To display the frames in a stack, an array is used.

JavaScript

```
function DisplayStack()
{
    // Create an array of stack frames in the current thread
    const Frames = Array.from(host.currentThread.Stack.Frames);
    host.diagnostics.debugLog("">>>> Printing stack \n");
    // Print out all of the frame entries in the array
    for(const [Idx, Frame] of Frames.entries())
    {
        host.diagnostics.debugLog("">>>> Stack Entry -> " + Idx + ":" + Frame
+ " \n");
    }
}
```

In this sample trace, the one stack entry is displayed.

dbgcmd

```
0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.DisplayStack()
>>> Printing stack
>>> Stack Entry -> 0: ntdll!LdrInitializeThunk + 0x21
```

Locating an event and displaying the stack

In this code all of the exceptions events are located and a loop is used to move to each one. Then the currentThread.ID of the [TTD Thread Objects](#) is used to display the thread ID and currentThread.Stack is used to display all of the frames in the stack.

JavaScript

```
function HardwareExceptionDisplayStack()
{
    var exceptionEvents = host.currentProcess.TTD.Events.Where(t => t.Type ==
"Exception");
    for (var curEvent of exceptionEvents)
    {
        // Move to the current event position
        curEvent.Position.SeekTo();
        host.diagnostics.debugLog("">>>> The Thread ID (TID) is : " +
host.currentThread.Id + "\n");
        // Create an array of stack frames in the current thread
        const Frames = Array.from(host.currentThread.Stack.Frames);
        host.diagnostics.debugLog("">>>> Printing stack \n");
        // Print out all of the frame entries in the array
        for(const [Idx, Frame] of Frames.entries()) {
            host.diagnostics.debugLog("">>>> Stack Entry -> " + Idx + ":" + Frame
+ " \n");
        }
    }
}
```

```

        }
        host.diagnostics.debugLog("\n");
    }
}

```

The output shows the location of the exception event, the TID and the stack frames.

```
dbgcmd

0:000> dx
Debugger.State.Scripts.MyTraceUtils.Contents.HardwareExceptionDisplayStack()
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: 91:0
>>> The Thread ID (TID) is : 5260
>>> Printing stack
>>> Stack Entry -> 0: 0x540020
>>> Stack Entry -> 1: 0x4d0049
>>> Stack Entry -> 2: DisplayGreeting!__CheckForDebuggerJustMyCode + 0x16d
>>> Stack Entry -> 3: DisplayGreeting!mainCRTStartup + 0x8
>>> Stack Entry -> 4: KERNEL32!BaseThreadInitThunk + 0x19
>>> Stack Entry -> 5: ntdll!_RtlUserThreadStart + 0x2f
>>> Stack Entry -> 6: ntdll!_RtlUserThreadStart + 0x1b
```

Locating an event and sending two commands

Querying TTD objects and sending commands can be combined as necessary. This example locates each event in the TTD trace of type ThreadCreated, moves to that position, and sends the [~ Thread Status](#) and the [!runaway](#) commands to display the thread status.

```
JavaScript

function ThreadCreateThreadStatus()
{
    var threadEvents = host.currentProcess.TTD.Events.Where(t => t.Type ==
"ThreadCreated");
    for (var curEvent of threadEvents)
    {
        // Move to the current event position
        curEvent.Position.SeekTo();
        // Display Information about threads
        host.namespace.Debugger.Utility.Control.ExecuteCommand("~", false);
        host.namespace.Debugger.Utility.Control.ExecuteCommand("!runaway 7",
false);
    }
}
```

Running the code displays the thread status at the moment in time that the exception occurred.

```
dbgcmd

0:000> dx
Debugger.State.Scripts.MyTraceUtils.Contents.ThreadCreateThreadStatus()
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: F:0
. 0 Id: 948.148c Suspend: 4096 Peb: 00a33000 Unfrozen
User Mode Time
    Thread      Time
    0:148c      0 days 0:00:00.000
Kernel Mode Time
    Thread      Time
    0:148c      0 days 0:00:00.000
Elapsed Time
    Thread      Time
    0:148c      3474 days 2:27:43.000
```

Chaining utility functions together

In this final sample, we can call the utility functions that we created earlier. First we index the trace using *IndexTraceTry* and then call *ThreadCreateThreadStatus*. We then use *ResetTrace* to move to the start of the trace and lastly call *HardwareExceptionDisplayStack*.

```
JavaScript

function ProcessTTDFiles()
{
    try
    {
        IndexTraceTry()
        ThreadCreateThreadStatus()
        ResetTrace()
        HardwareExceptionDisplayStack()
    }

    catch(err)
    {
        host.diagnostics.debugLog("\n >>> Processing of TTD file failed
\n");
    }
}
```

Running this script on a trace file that contains a hardware exception, generates this output.

```
dbgcmd

0:000> dx Debugger.State.Scripts.MyTraceUtils.Contents.ProcessTTDFiles()

>>> Index Return Value: Loaded

>>> Trace was successfully indexed in 0 ms
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: F:0
. 0 Id: 948.148c Suspend: 4096 Peb: 00a33000 Unfrozen
User Mode Time
    Thread      Time
    0:148c      0 days 0:00:00.000
Kernel Mode Time
    Thread      Time
    0:148c      0 days 0:00:00.000
Elapsed Time
    Thread      Time
    0:148c      3474 days 2:27:43.000
>>> Printing stack
>>> Stack Entry -> 0: ntdll!LdrInitializeThunk
>>> Current position in trace file: F:0
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: F:0
>>> New position in trace file: F:0
(948.148c): Break instruction exception - code 80000003 (first/second chance
not available)
Time Travel Position: 91:0
>>> The Thread ID (TID) is : 5260
>>> Printing stack
>>> Stack Entry -> 0: 0x540020
>>> Stack Entry -> 1: 0x4d0049
>>> Stack Entry -> 2: DisplayGreeting!__CheckForDebuggerJustMyCode + 0x16d
>>> Stack Entry -> 3: DisplayGreeting!mainCRTStartup + 0x8
>>> Stack Entry -> 4: KERNEL32!BaseThreadInitThunk + 0x19
>>> Stack Entry -> 5: ntdll!_RtlUserThreadStart + 0x2f
>>> Stack Entry -> 6: ntdll!_RtlUserThreadStart + 0x1b
```

See Also

[Time Travel Debugging - Overview](#)

[Introduction to Time Travel Debugging objects](#)

[JavaScript Debugger Scripting](#)

[JavaScript Debugger Example Scripts](#)

Feedback

Was this page helpful?

 Yes

 No

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

Time Travel Debugging - TTD.exe command line utility

Article • 11/15/2024



This article describes when and how to use the TTD.exe command line utility to record a trace.

When to use the TTD.exe command line utility

Time Travel Debugging (TTD) allows you to record the code execution of an app or process and save it in a trace file. The file can be played back in the Windows debugger to locate an issue with code execution.

For many scenarios, the easiest way to use TTD to record an app or process is directly from the WinDbg UI. For information on time travel debugging using the WinDbg UI, see [Time Travel Debugging - Overview](#).

You may have scenarios where only the TTD command line recorder is required: recording on a PC without installing the debugger, advanced recording scenarios, test automation, etc. In these scenarios you can install just the TTD command line recorder through a URL.

TTD recording impacts the recorded process

TTD recording is an invasive technology. You will notice anywhere from 5x-20x or more slowdown of the running app or process while recording, depending on the application and the recording options selected.

The created trace files grow over time and can take significant storage space. Work to trace for the shortest period of time, capturing the program activity of interest and then close the trace as soon as possible.

Once TTD is attached to a process, it cannot be removed from it. Close the app or end the process once TTD recording is complete. For system-critical processes, this will require a reboot of the OS.

TTD recordings may contain personally identifiable or security related information

TTD recordings capture memory contents and may contain personally identifiable or security related information, including but not necessarily limited to file paths, registry, memory or file contents. The exact information depends on target process activity while it was recorded.

How to download and install the TTD.exe command line utility (Preferred method)

Download the TTD command line utility here - <https://aka.ms/ttd/download> ↗

Select *Install* and TTD will download and install. The TTD command is added to the system path and is available for use at the command prompt, when the install completes.

If you encounter difficulties installing, see [Troubleshoot installation issues with the App Installer file](#).

On some PC's you may need to install the [Microsoft App Installer for Windows 10](#) ↗. It is available in the Microsoft Store app in Windows. Windows Package Manager is supported through App Installer starting on Windows 10 1809.

How to download and install the TTD.exe command line utility (Offline method)

While the preferred installation method is to use the App Installer, you can also download the TTD command line package and extract the files manually. Here are two ways to do it.

Extract the files from an already installed TTD.exe command line utility

If you have already installed the TTD command line utility, you can extract the files from the installed location. In Powershell you would do this to find the installed location:

```
PowerShell
```

```
(Get-AppxPackage | where Name -eq
```

```
'Microsoft.TimeTravelDebugging').InstallLocation
```

From there you can copy all the binaries (*.dll, *.exe, *.sys) to a new location. Here is one way to do this in Powershell:

PowerShell

```
robocopy.exe (Get-AppxPackage | where Name -eq  
'Microsoft.TimeTravelDebugging').InstallLocation c:\myttd *.exe *.dll *.sys  
/E /XD AppxMetadata
```

Replace "c:\myttd" with the destination of your choice. The result will look something like this (on an x64 machine):

Console

```
ls -Recurse c:\myttd
```

```
Directory: C:\myttd
```

Mode	LastWriteTime	Length	Name
-	-	-	x86
d----	11/9/2023 2:43 PM	79240	ProcLaunchMon.sys
-a---	11/9/2023 2:43 PM	112568	TTD.exe
-a---	11/9/2023 2:43 PM	309176	TTDInject.exe
-a---	11/9/2023 2:43 PM	55328	TTDLoader.dll
-a---	11/9/2023 2:43 PM	821176	TTDRecord.dll
-a---	11/9/2023 2:43 PM	1222584	TTDRecordCPU.dll
-a---	11/9/2023 2:43 PM	63416	TTDRecordUI.dll

```
Directory: C:\myttd\x86
```

Mode	LastWriteTime	Length	Name
-	-	-	-
-a---	11/9/2023 2:43 PM	247728	TTDInject.exe
-a---	11/9/2023 2:43 PM	42928	TTDLoader.dll
-a---	11/9/2023 2:43 PM	1128480	TTDRecordCPU.dll

Note that the x86 binaries are in a subdirectory. If you do not need to record 32-bit processes this folder can be deleted (and you could add /xd x86 to the robocopy command to avoid copying it in the first place). The ARM64 version doesn't have any subdirectories.

The TTDRecordUI.dll is only needed if you want to use the UI to control recording. If you do not want the UI, you can delete this file.

Download the TTD.exe command line utility package and extract the files manually

If you do not want to install the TTD command line utility, you can download the package and extract the files manually. The following Powershell script will:

- Get the URL for the current version of TTD from <https://aka.ms/ttd/download>.
- Download the MSIX bundle.
- Extract the requested architecture's MSIX from MSIX bundle.
- Extract the TTD binaries from the MSIX.

```
PowerShell

param(
    $OutDir = ".",
    [ValidateSet("x64", "x86", "arm64")]
    $Arch = "x64"
)

# Ensure the output directory exists
if (!(Test-Path $OutDir)) {
    $null = mkdir $OutDir
}

# Ensure the temp directory exists
$TempDir = Join-Path $OutDir "TempTtd"
if (!(Test-Path $TempDir)) {
    $null = mkdir $TempDir
}

# Determine if the destination already contains binaries
$extensions = @('.dll', '.exe', '.sys')
$existingBinaries = (Get-ChildItem -recurse $OutDir | Where-Object Extension -In $extensions).Count -gt 0

# Download the appinstaller to find the current uri for the msixbundle
Invoke-WebRequest https://aka.ms/ttd/download -OutFile
$TempDir\ttd.appinstaller

# Download the msixbundle
$msixBundleUri = ([xml](Get-Content
$TempDir\ttd.appinstaller)).AppInstaller.MainBundle.Uri

if ($PSVersionTable.PSVersion.Major -lt 6) {
    # This is a workaround to get better performance on older versions of
    # PowerShell
    $ProgressPreference = 'SilentlyContinue'
}

# Download the msixbundle (but name as zip for older versions of Expand-
Archive)
```

```

Invoke-WebRequest $msixBundleUri -OutFile $TempDir\ttd.zip

# Extract the 3 msix files (plus other files)
Expand-Archive -DestinationPath $TempDir\UnzippedBundle $TempDir\ttd.zip -Force

# Expand the build you want - also renaming the msix to zip for Windows
PowerShell
$fileName = switch ($Arch) {
    "x64" { "TTD-x64" }
    "x86" { "TTD-x86" }
    "arm64" { "TTD-ARM64" }
}

# Rename msix (for older versions of Expand-Archive) and extract the debugger
Rename-Item "$TempDir\UnzippedBundle\$fileName.msix" "$fileName.zip"
Expand-Archive -DestinationPath "$OutDir"
"$TempDir\UnzippedBundle\$fileName.zip"

# Delete the temp directory
Remove-Item $TempDir -Recurse -Force

# Remove unnecessary files, if it is safe to do so
if (-not $existingBinaries) {
    Get-ChildItem -Recurse -File $OutDir |
        Where-Object Extension -NotIn $extensions |
        Remove-Item -Force

    Remove-Item -Recurse -Force (Join-Path $OutDir "AppxMetadata")
} else {
    Write-Host "Detected pre-existing binaries in '$OutDir' so did not remove any files from TTD package."
}

```

Assuming you saved the above script as `Get-Ttd.ps1`, you can run it like this to download the x64 binaries to the `c:\myttd` directory:

PowerShell

```

md c:\myttd
cd c:\myttd
.\Get-Ttd.ps1

```

Or you could specify the output directory and architecture:

PowerShell

```
.\Get-Ttd.ps1 -OutDir c:\myttd-arm64 -Arch arm64
```

Replace "c:\myttd" or "c:\myttd-arm64" with the destination of your choice.

How to record a trace using the TTD.exe command line utility

There are three ways to record a trace.

- Launch a process
- Attach to a process
- Monitor a process

Once the process is being recorded, you need to trigger the issue that you wish to debug. You might open a problematic file or click on a specific button in the app to cause the event of interest to occur. When the application being recorded terminates, naturally or by crashing, the trace file will be finalized.

Tip

Recording TTD traces requires administrative rights. Typically this is done by running ttd.exe from an administrator command prompt.

For more information about recording a time travel trace using WinDbg, see [Time Travel Debugging - Record a trace](#).

Launch a process

```
-launch <Program> [<arguments>]
```

Launch and trace the program (default mode).

This is the only mode that allows you to pass arguments to the program. The program will launch with the same privileges as TTD.exe (as an admin). Use `-attach` or `-monitor` to record the program with its normal set of privileges.

Including `-launch` is optional, but may be used for clarity.

The first unrecognized argument that doesn't start with - or / will be assumed to be an executable path to launch, and any subsequent arguments will be assumed to be the arguments for that program.

For example, use `TTD.exe notepad.exe` to launch and record notepad. The trace will stop when you close notepad.

For example usage, see [Scenario usage examples - recording a process](#).

Attach to a process

`-attach <PID>`

Attach to a running process specified by process ID. Use TaskManager or the TaskList utility to identify process numbers. For more information, see [Finding the process ID](#).

For example, use `TTD.exe -attach 21440 -out C:\traces\MyTraceFile.run` to launch and record the process with an ID of 21440 and save the trace in MyTraceFile.run.

Make sure the directory exists (`C:\traces` in this example) before running TTD.exe.

For example usage, see [Scenario - Locate and attach to a running process](#).

Monitor a process

`-monitor <Program>`

The monitor option allows for a program to be monitored and traced each time they are started. To use this option, you must specify a full path to the output location with `-out`.

To stop monitoring press Ctrl+C.

The main benefits of monitoring compared to the other methods are:

- You can launch the target app the normal way, no need to figure out the command line to start it.
- The target app will run with its normal privileges. If you launch the app directly from ttd.exe it will launch elevated and that may change the behavior of the program.
- It is useful for automation (use a script that monitors the launch of a program and collects a trace).

The `-monitor` option can be specified more than once to monitor multiple programs.

For example usage, see [Scenario usage examples - monitoring processes](#).

Command line options

Syntax

```
TTD.exe [options] [mode] [program [<arguments>]]
```

```
-? | -help
```

Display the command line help.

Modes

```
-launch <Program> [<arguments>]
```

Launch and trace the program (default mode).

This is the only mode that allows you to pass arguments to the program. The `-launch` option must be the last TTD option in the command-line, followed by the program to launch, and any arguments that the program requires. If no mode is specified it will be treated as launch as well. For example `TTD.exe -out C:\traces ping.exe msn.com` is treated as a launch.

```
-attach <PID>
```

Attach to a running process specified by process ID. Use TaskManager or TaskList utility to identify process IDs. For more information, see [Finding the process ID](#).

```
-monitor <Program>
```

Trace programs or services each time they are started (until reboot). To use this option, you must specify a full path to the output location with `-out`.

Basic command line options

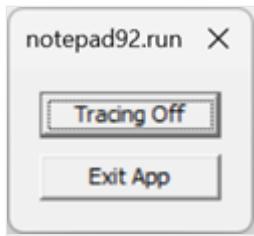
```
-out <path>
```

Specify a trace file name or a directory. If a directory, the directory must already exist. If a file name, the file name must not exist.

```
-noUI
```

Disables the UI for manual control of recording.

If this option is not selected a small UI is displayed when recording is active. "Tracing Off" stops tracing and app continues; "Exit App" closes the app which also stops tracing.



`-accepteula`

Use this option to accept the EULA user license agreement. This option can be used in automation scenarios, after the EULA has been reviewed and accepted.

TTD displays the EULA the first time it is run. Type Y or N to accept the EULA. Once accepted, the EULA will no longer be displayed at startup. If the EULA is not accepted TTD exits, and the EULA will be displayed, the next time TTD is run.

Trace control

`-stop <process name> | <PID> | all`

Stop tracing the specified process name, PID or "all" can be specified.

`-wait <timeout>`

Wait for up to the amount of seconds specified for all trace sessions on the system to end. Specify -1 to wait infinitely.

`-tracingOff`

Starts application with trace recording off. You can use the UI checkbox to turn tracing back on after it has been turned off.

Additional command line options

`-children`

Record the target as well as any processes created by the target. Each child process will be recorded into its own trace file.

`-cmdLineFilter "<string>"`

Record the target if its command line contains the string. This option works only with `-monitor` mode. It is useful for situations when the command line argument uniquely identifies the process you are interested in. For example, `-monitor notepad.exe -`

`cmdLineFilter "specialfile.txt"` records notepad.exe only if specialfile.txt appears on the command line.

`-cleanup`

Uninstall process monitor driver.

Trace behavior settings

`-timestampFilename`

Adds a timestamp to the last part of the trace file name. For example, ping_2023-06-17_103116.run.

For example to record ping.exe, with a timestamp included in the file name, use this command.

Console

```
ttd.exe -out c:\traces -timestampFilename ping.exe msn.com
```

By default a sequential scan is done to find an unused file in the output directory. If ping.exe is recorded the recorder will try ping01.run, ping02.run, etc. until an unused file name is found. For most scenarios this naming method is sufficient. However, if you want to record the same program many times, the default file naming algorithm can become inefficient, when there is large number of existing files.

`-ring`

Trace to a ring buffer. The file size will not grow beyond the limits specified by `-maxFile`. Only the last portion of the recording that fits within the given size, will be saved.

`-maxFile <size>`

Maximum size of the trace file in MB. When in full trace mode the default is 1024GB and the minimum value is 1MB. When in ring buffer mode the default is 2048MB, the minimum value is 1MB, and the maximum value is 32768MB.

The default for in-memory ring on 32-bit processes is 256MB.

`-maxConcurrentRecordings <count>`

Maximum number of recordings that can be ongoing at any one point in time. If not specified, an unlimited number of recordings can occur simultaneously.

`-numVCpu <number>`

Specifies a number of Virtual CPUs to be reserved and used when tracing. This value affects the total memory overhead placed on the guest process' memory by TTD. If not specified then default per platform is: 55 for x64/ARM64 and 32 for x86.

Change this setting in order to limit the memory impact *only* if you are running out of memory. Changing the numVCpu value to a lower number can severely impact the performance of tracing and should only be done to work around memory usage issues.

If TTD.exe fails to record, or the .out file indicates a simulation of 0 seconds, using `-numVCpu` may enable the recording to succeed.

`-replayCpuSupport <support>`

Specifies what support is expected from the CPUs that will be used to replay the trace. The default setting is recommended for portability of traces between machines but other options may be used to produce small traces files and record faster (depending on the specific instructions used by the target program).

`<support>` values

[\[+\] Expand table](#)

Value	Description
<code>Default</code>	Default CPU support, just requires basic commonly-available support in the replay CPU.
<code>MostConservative</code>	Requires no special support in the replay CPU. Adequate for traces that will be replayed on a completely different CPU architecture, like an Intel trace on ARM64 CPU.
<code>MostAggressive</code>	Assumes that the replay CPU will be similar and of equal or greater capability than the CPU used to record.
<code>IntelAvxRequired</code>	Assumes that the replay CPU will be Intel/AMD 64-bit CPU supporting AVX.
<code>IntelAvx2Required</code>	Assumes that the replay CPU will be Intel/AMD 64-bit CPU supporting AVX2.

Reducing overhead of tracing

While TTD is very efficient for what it does (full instruction level tracing encoded into less than one byte/instruction on average), it still has noticeable overhead when recording. Modern CPUs can execute billions of instructions per second, making even

one byte/instruction expensive. In many cases recording the entire process is not necessary.

The following options can be used to reduce the overhead of tracing:

```
-module <module name>
```

Record only the specified module (such as comdlg32.dll) and the code that it calls. This can be the executable itself or any DLL loaded by the executable. This option can be specified more than once to record multiple modules.

When this option is used the target process runs at full speed until code in the specified module(s) is executed. TTD will then record the process until execution leaves the specified module(s), at which point recording is turned off and the target returns to full speed. Because turning recording on/off is expensive, TTD will leave recording on when a specified module calls out to other modules in the process.

```
-recordmode <Automatic | Manual>
```

Normally recording starts as soon as TTD injects itself in the target process ("Automatic" mode, the default). If your program makes use of TTD's [in-process recording API](#) to control when recording occurs then you can use the "Manual" mode to run at full speed until your program calls the API to start recording.

Using these options can result in a significant reduction in recording overhead and trace file size. Debugging a trace recorded with these options is no different than a trace of the entire process. Whenever you reach a location in the trace where recording is turned off, the next instruction in the trace is the first instruction executed when recording resumed.

Event related settings

```
-passThroughExit
```

Pass the guest process exit value through as TTD.exe's exit value. This value is available to batch files through the `%ERRORLEVEL%` variable. Powershell and other command line environments offer mechanisms for getting the process exit value as well.

```
-onInitCompleteEvent <eventName>
```

Allows an event to be signaled when tracing initialization is complete.

Scenario usage examples - recording a process

Scenario - Launch and record a Windows app

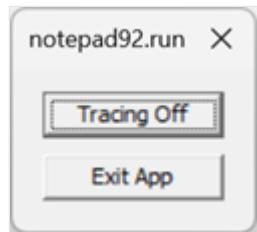
In this scenario notepad is launched and a trace is created.

1. Use the `-launch` option to start notepad and record it.

```
Console

C:\TTD> TTD.exe -launch notepad.exe
Launching 'notepad.exe'
    Recording process (PID:9960) on trace file: C:\TTD\notepad01.run
notepad.exe(x64) (PID:9960): Process exited with exit code 0 after 12984ms
    Full trace dumped to C:\TTD\notepad01.run
```

2. A small application menu is displayed showing that tracing is on.



3. When the application is closed, a trace file is generated. In this example `notepad01.run`.

Scenario - Launch and record a Windows app with a passed parameter

In this scenario ping is started, and the address to ping is passed in as a parameter.

1. In this example the `-launch` option is omitted as that is the default mode.

```
Console

C:\TTD> TTD.exe ping.exe msn.com
Launching 'ping.exe msn.com'
    Recording process (PID:24044) on trace file: C:\TTD\ping01.run

Pinging msn.com [204.79.197.219] with 32 bytes of data:
Reply from 204.79.197.219: bytes=32 time=22ms TTL=118
Reply from 204.79.197.219: bytes=32 time=21ms TTL=118
Reply from 204.79.197.219: bytes=32 time=25ms TTL=118
Reply from 204.79.197.219: bytes=32 time=21ms TTL=118

Ping statistics for 204.79.197.219:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
```

```
Minimum = 21ms, Maximum = 25ms, Average = 22ms
ping.exe(x64) (PID:24044): Process exited with exit code 0 after 3390ms
Full trace dumped to C:\TTD\ping01.run
```

- When the application is closed, a trace file is generated. In this example ping01.run.

Scenario - Locate and attach to a running process

In this scenario notepad is started, its process ID is located and a trace is created by attaching to the running application

- Start the target app, in this example notepad.
- Use TaskList or other methods described in to locate the process ID. For more information, see [Finding the process ID](#).

```
Console

C:\TTD> TaskList
...
Notepad.exe           21440 Console                 1      73,020 K
...
```

- Using that process ID, use the `-attach` option to attach and record it. Optionally specify a filename for the trace file using `-out`.

```
Console

C:\TTD> TTD.exe -attach 21440 -out C:\TTD\MyTraceFile.run
Attaching to 21440
Recording process (PID:21440) on trace file: C:\TTD\MyTraceFile.run
(x64) (PID:21440): Process exited with exit code 0 after 26672ms
Full trace dumped to C:\TTD\MyTraceFile.run
```

Scenario - Recording a parent and its children processes

In this scenario a parent and its children processes will be recorded. As some apps may use many children processes, the family trace file that contains the children, may become quite large.

- Specify the `-children` option and the name of the parent app to record.

This is an example of recording cmd.exe launching ping.exe as a child process.

Console

```
ttd.exe -out d:\traces -children cmd.exe /C ping.exe msn.com

Microsoft (R) TTD 1.01.11
Release: 1.11.0.0
Copyright (C) Microsoft Corporation. All rights reserved.

Launching 'cmd.exe /C ping.exe msn.com'
Recording process (PID:48200) on trace file: d:\traces\cmd01.run
Recording process (PID:53724) on trace file: d:\traces\PING01.run

Pinging msn.com [204.79.197.219] with 32 bytes of data:
Reply from 204.79.197.219: bytes=32 time=6ms TTL=117
Reply from 204.79.197.219: bytes=32 time=6ms TTL=117
Reply from 204.79.197.219: bytes=32 time=7ms TTL=117
Reply from 204.79.197.219: bytes=32 time=7ms TTL=117

Ping statistics for 204.79.197.219:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 6ms, Maximum = 7ms, Average = 6ms
(x64) (PID:53724): Process exited with exit code 0 after 3516ms
    Trace family nesting level is 1; Parent process ID is 48200
    Full trace dumped to d:\traces\PING01.run

...
```

2. Multiple trace files are created: one for the parent process and a trace file for each child process. WinDbg only opens one trace file at a time so you will need to run separate instances of WinDbg for each trace, if you want to debug them at the same time.

Scenario usage examples - monitoring processes

Scenario - monitoring for program launches and starting recording

In this scenario the `-monitor` option is used to record all currently running instances as well as future instances of notepad.exe, until system is rebooted or ttd.exe is exited via Ctrl+C. The `-out` option is required for monitor, and the output folder must exist already.

1. Monitor and trace the current, as well as any future instances of notepad.exe.

Console

```
C:\TTD> TTD.exe -out C:\TTD\ -monitor notepad.exe
Microsoft (R) TTD 1.01.11
Release: 1.11.121.0
Copyright (C) Microsoft Corporation. All rights reserved.

The Process Launch Monitor driver is not installed
Successfully installed the Process Launch Monitor driver
Recording process Notepad.exe(15904)           From parent process
explorer.exe(8440)
    Recording process (PID:15904) on trace file: C:\TTD\notepad01.run
Recording process Notepad.exe(19920)           From parent process
explorer.exe(8440)
    Recording process (PID:19920) on trace file: C:\TTD\notepad02.run
(x64) (PID:19920): Process exited with exit code 0 after 1281ms
    Full trace dumped to C:\TTD\notepad02.run

(x64) (PID:15904): Process exited with exit code 0 after 30719ms
    Full trace dumped to C:\TTD\notepad01.run
```

2. In this example two instances of notepad.exe were loaded after tracing had started.

After the activity of interest was captured, CTRL-C, was used at the command prompt to stop the recording.

Scenario - monitoring two programs for program launches

In this scenario the `-monitor` option is used to monitor and record two applications.

1. Monitor and trace the current, as well as any future instances of notepad.exe and ping.exe.

Console

```
C:\TTD> TTD.exe -out C:\TTD\ -monitor notepad.exe -monitor ping.exe
Microsoft (R) TTD 1.01.11
Release: 1.11.121.0
Copyright (C) Microsoft Corporation. All rights reserved.

Successfully uninstalled the Process Launch Monitor driver
Successfully installed the Process Launch Monitor driver
Recording process Notepad.exe(17972)           From parent process
explorer.exe(8440)
    Recording process (PID:17972) on trace file: C:\TTD\Notepad01.run
Tracking process svchost.exe(7824)           From parent process
```

```

services.exe(1292)
Tracking process sppsvc.exe(10376)           From parent process
services.exe(1292)
Tracking process ClipUp.exe(15108)            From parent process
svchost.exe(7824)
Tracking process ClipUp.exe(21180)            From parent process
ClipUp.exe(15108)
Tracking process consent.exe(24280)           From parent process
svchost.exe(892)
Tracking process ctfmon.exe(24508)           From parent process
svchost.exe(5064)
Tracking process wt.exe(10768)                 From parent process explorer.exe(8440)
Tracking process WindowsTerminal.exe(23296)    From parent process
wt.exe(10768)
Tracking process OpenConsole.exe(6816)          From parent process
WindowsTerminal.exe(23296)
Tracking process powershell.exe(15956)         From parent process
WindowsTerminal.exe(23296)
Tracking process git.exe(3656)                 From parent process
powershell.exe(15956)
Tracking process git.exe(1928)                 From parent process git.exe(3656)
Tracking process git.exe(20312)                 From parent process
powershell.exe(15956)
Tracking process git.exe(5712)                 From parent process git.exe(20312)
Tracking process csc.exe(16144)                From parent process
powershell.exe(15956)
Tracking process cvtres.exe(19488)             From parent process csc.exe(16144)
Recording process PING.EXE(21468)              From parent process
powershell.exe(15956)

        Recording process (PID:21468) on trace file: C:\TTD\PING01.run
(x64) (PID:21468): Process exited with exit code 1 after 234ms
        Full trace dumped to C:\TTD\PING01.run

```

```

Tracking process Teams.exe(10060)             From parent process Teams.exe(2404)
Tracking process cmd.exe(21796)                From parent process
powershell.exe(15956)
Recording process PING.EXE(364)               From parent process cmd.exe(21796)
        Recording process (PID:364) on trace file: C:\TTD\PING02.run
(x64) (PID:364): Process exited with exit code 1 after 234ms
        Full trace dumped to C:\TTD\PING02.run

```

2. In this example notepad.exe and then ping.exe were loaded after tracing had started. After the activity of interest was captured, CTRL-C, was used at the command prompt to stop the recording.

Scenario - Stopping the recording in a second window

In this scenario the activity of interest was captured, and all recording is stopped using `-stop all`. A second command window is used to execute the `-stop all` option.

Console

```
C:\TTD> TTD.exe -stop all
Microsoft (R) TTD 1.01.11
Release: 1.11.121.0
Copyright (C) Microsoft Corporation. All rights reserved.

Full trace written to 'C:\TTD\Notepad01.run'
```

Scenario - Cleaning up the monitor driver

In this scenario the `-cleanup` option is used to clean up the monitor driver after all recording is complete.

Console

```
C:\TTD> TTD.exe -cleanup
The monitor service is not installed
Successfully uninstalled the Process Launch Monitor driver
```

Additional command line examples

This table highlights some additional command line usage examples. Refer to the [Command line options](#) for additional information about the illustrated options.

[+] Expand table

Scenario	Command	Description
Attach to process but do not start recording yet	<code>Ttd.exe -tracingoff</code> <code>notepad.exe</code>	Launches notepad with recording turned off. The recording can be started at any time through the UI.
Filter by command line	<code>Ttd.exe -cmdlinefilter</code> <code>foo.txt -monitor</code> <code>notepad.exe</code>	Record notepad.exe but only if foo.txt is on the command line when it is launched, placing output in current directory.
Ring recording	<code>Ttd.exe -ring -attach 1234</code>	Records PID 1234 into a trace file capped to 2GB, placing output in the current directory. Older contents in the trace file are overwritten as needed to keep the file under the maximum size. Use <code>-maxfile</code> to change the max size.

Scenario	Command	Description
Limit trace file size	<code>Ttd.exe -maxfile 4096</code> <code>notepad.exe</code>	Record notepad.exe until the trace file reaches 4GB, placing output in current directory.
Limit number of recordings that happen at same time	<code>Ttd.exe -maxconcurrentrecordings 1</code> <code>-out c:\my\dir -monitor</code> <code>notepad.exe</code>	Recording is CPU-intensive and in some cases.
Reduce memory usage in the target process	<code>Ttd.exe -numvcpu 8 -monitor</code> <code>w3wp.exe</code>	Some processes, such as w3wp.exe, set a small quota on the amount of memory it can use. If ttd.exe fails to start recording, use <code>-numvcpu</code> to reduce the number of virtual CPUs TTD allocates. Only try this option if ttd.exe is unable to record through other means.
Choose between trace portability and recording speed / trace file size	<code>Ttd.exe -replaycpusupport</code> <code>mostaggressive</code> <code>notepad.exe</code>	By default TTD produces trace files that are portable across a wide range of hardware. Choosing 'mostaggressive' tells TTD it is ok to record a trace which can only be played back on CPUs with the same capabilities as the machine that recorded the trace. In some cases this can substantially improve recording speed and trace file size.

Automation command line examples

This table highlights some additional command line usage examples that can be useful for the automated use of TTD.exe utility.

[\[+\] Expand table](#)

Scenario	Command	Description
Disable UI	<code>Ttd.exe -noui -accepteula</code> <code>notepad.exe</code>	Record notepad.exe, placing output into current directory, without showing the UI.
Wait for recorder to start programmatically	<code>Ttd.exe -accepteula -oninitcompleteevent</code> <code>ttd_notepad</code> <code>notepad.exe</code>	Create a Win32 named event 'ttd_notepad' and launch notepad.exe. TTD will signal 'ttd_notepad' when recording is initialized. Automation can wait on the event before proceeding with the behavior they want to record.
Preserve target's exit code	<code>Ttd.exe -accepteula -passthroughexit</code> <code>ping.exe</code>	Records ping.exe, placing output in current directory. Ttd.exe's exit code will be the same as ping.exe's exit code.

Scenario	Command	Description
	msn.com	
Wait for recording to end	Ttd.exe -accepteula -wait 30	After recording is stopped, wait up to 30 seconds for TTD to finish writing trace file to disk. Use <code>-wait -1</code> to wait indefinitely.

All of these examples use the `-accepteula` option to make sure that the automation is not blocked by the EULA confirmation dialog.

Working with the generated trace file

For information on working with a trace and instructions on how to replay time travel traces, and navigate forwards and backwards in time, see [Time Travel Debugging - Replay a trace](#).

Tips on working with trace files

- When sharing traces with others you only have to share the .run file. The index file (.idx) can be as large as the .run file and is automatically created when the trace file is loaded by WinDbg.
- When collaborating with others, pass on any relevant trace positions related to the problem at hand. The collaborator can use the `!tt x:y` command to move to that exact point in time in the execution of the code. Time position ranges can be included in bug descriptions to track where the possible issue may be occurring.
- When reporting an issue with TTD, if you supply the .run file, supply the .out file as well. This allows for the confirmation that the recording process worked properly.
- Trace files (.run) compress well.

Troubleshooting TTD.exe

There are some cases where trace file errors can occur. For more information, see [Time Travel Debugging - Troubleshooting](#).

The .out file can be used for troubleshooting. The example out file shows a functional trace, ending with an exit code of zero.

```
Console

Microsoft (R) TTDRecord 1.01.11
Release: 1.11.47.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Initializing Time Travel Debugging for Attach to 9916
Time: 05/08/2023 17:07:15
OS:10.0.25314 EDITION:x64

SessionID: 008F918C-B8A7-4C4E-B91B-34CFC953C501

        (TTD::ManageTTDTrace:2725)
Running
        (TTD::StartGuestProcess:1512)
Group tracing GUID: B84DF180-DA54-46E5-9019-73690C689979

Running "C:\WINDOWS\SYSTEM32\TTDInject.exe" /duration 1 /InjectMode
LoaderForCombinedRecording /ClientParams "37 C:\TTD\Notepad03.run 0 0 0 0 0
0 0 0 c06001 0" /RecordScenario 268435458 /attach 9916 -TraceFileHandle 4f8
-GuestEventHandle 380 -ClientEventHandle 384 -ActiveEventHandle 4f4 -
MutexHandle 46c -CommunicationBufferHandle 3c0 -SharedSequenceMutexHandle
3b8 -SharedSequenceBufferHandle 330 /TelemetryFeatureSessionId "008F918C-
B8A7-4C4E-B91B-34CFC953C501"
        (TTD::StartGuestProcess:1955)
Microsoft (R) TTDInject 1.01.11
Release: 1.11.27.0
Copyright (C) Microsoft Corporation. All rights reserved.

TTDLoader Params:
LauncherDll = TTDLoader
ClientDll   = TTDRRecordCPU
ClientEntry = InitializeNirvanaClient
ClientParams= 37 C:\TTD\Notepad03.run 0 0 0 0 0 0 0 0 c06001 0
Attach
WaitForMain is off
Allocated processors:55, running threads:2.
Loader TTDLoader.dll injected at 0x00007FFF423B0000 0xc000 -- .reload
TTDLoader.dll=0x00007FFF423B0000,0xc000

Injection by thread is complete.
RecordingEngine initialization successful.
RecordVcpu initialization successful.
Loader initialization successful.
Guest Process is x64 binary.
Tracing started at: Tue May  9 00:07:16 2023 (UTC) Mon May  8 17:07:16 2023
(Local)

Guest process exited with exit code 0
Simulation time of '' (x64): 18781ms.
Tracing completed at: Tue May  9 00:07:34 2023 (UTC) Mon May  8 17:07:34
2023 (Local)
```

Most of the .out file content is used internally by the time travel debugging team to troubleshoot recording errors. The following information can be helpful to others that are working with the trace file.

- Some error messages are only displayed in the .out file and may be used to determine the specifics of the failure.
- Indication of wall clock time the recording started / stopped
- How long the recording session lasted (simulation time)
- Whether the recording is a launch (with command line) or attach recording
- The OS version

See Also

[Time Travel Debugging - Overview](#)

[Time Travel Debugging - Record a trace](#)

[Time Travel Debugging - Replay a trace](#)

Feedback

Was this page helpful?



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

Using Debugger Commands

Article • 10/25/2023

This describes the use of debugger commands. WinDbg is a debugger that can be used to analyze crash dumps, debug live user-mode and kernel-mode code, and examine CPU registers and memory. For more information, see [WinDbg Overview](#).

To install the debugger, see [Install the Windows debugger](#).

To get started with WinDbg, see [Getting Started with Windows Debugging](#).

WinDbg Debugger Command window

For WinDbg, "Debugger Command window" refers to the window that is labeled "Command" in the title bar. This window contains two panes:

- In the small, bottom pane, you enter commands.
- In the large, upper pane, you view command output.

This window is always open at the beginning of a debugging session. You can reopen or switch to this window by selecting **Command** on the **View** menu, pressing ALT+1, or selecting the **Command (Alt+1)** button () on the toolbar.

You can use the UP ARROW and DOWN ARROW keys to scroll through the command history. When a previous command appears, you can edit it and then press ENTER to execute the previous command (or the edited version of the previous command). The cursor does not have to be at the end of the line for this procedure to work correctly.

KD or CDB

For KD or CDB, "Debugger Command window" refers to the whole window. You enter commands at the prompt at the bottom of the window. If the commands have any output, the window displays the output and then displays the prompt again.

Debugger Command Window Prompt

When you are performing user-mode debugging, the prompt in the Debugger Command window looks like the following example.

2:005>

In the preceding example, 2 is the current process number, and 005 is the current thread number.

If you attach the debugger to more than one computer, the system number is included before the process and thread number, as in the following example.

```
3:2:005>
```

In this example, 3 is the current system number, 2 is the current process number, and 005 is the current thread number.

When you are performing kernel-mode debugging on a target computer that has only one processor, the prompt looks like the following example.

```
kd>
```

However, if the target computer has multiple processors, the number of the current processor appears before the prompt, as in the following example.

```
0: kd>
```

If the debugger is busy processing a previously issued command, new commands will temporarily not be processed, although they can be added to the command buffer. In addition, you can still use [control keys](#) in KD and CDB, and you can still use menu commands and [shortcut keys](#) in WinDbg. When KD or CDB is in this busy state, no prompt is displayed. When WinDbg is in this busy state, the following indicator will appear in place of the prompt:

```
*BUSY*
```

You can use the [.pcmd \(Set Prompt Command\)](#) command to add text to this prompt.

Kinds of Commands

WinDbg, KD, and CDB support a variety of commands. Some commands are shared between the debuggers, and some are available only on one or two of the debuggers.

Some commands are available only in live debugging, and other commands are available only when you debug a dump file.

Some commands are available only during user-mode debugging, and other commands are available only during kernel-mode debugging.

Some commands are available only when the target is running on certain processors. For more information about all of the commands and their restrictions, see [Debugger](#)

Commands.

Editing, Repeating, and Canceling Commands

You can use standard editing keys when you enter a command:

- Use the UP ARROW and DOWN ARROW keys to find previous commands.
- Edit the current command with the BACKSPACE, DELETE, INSERT, and LEFT ARROW and RIGHT ARROW keys.
- Press the ESC key to clear the current line.

You can press the TAB key to automatically complete your text entry. In any of the debuggers, press the TAB key after you enter at least one character to automatically complete a command. Press the TAB key repeatedly to cycle through text completion options, and hold down the SHIFT key and press TAB to cycle backward. You can also use wildcard characters in the text and press TAB to expand to the full set of text completion options. For example, if you type `fo*!ba` and then press TAB, the debugger expands to the set of all symbols that start with "ba", in all modules with module names that start with "fo". As another example, you can complete all extension commands that have "prcb" in them by typing `!*prcb` and then pressing TAB.

When you use the TAB key to perform text completion, if your text fragment begins with a period (.), the text is matched to a dot command. If your text fragment begins with an exclamation point (!), the text is matched to an extension command. Otherwise, the text is matched with a symbol. When you use the TAB key to enter symbols, pressing the TAB key completes code and type symbols and module names. If no module name is apparent, local symbols and module names are completed. If a module or module pattern is given, symbol completion completes code and type symbols from all matches.

You can select and hold (or right-click) in the Debugger Command window to automatically paste the contents of the clipboard into the command that you are typing.

The maximum command length is 4096 characters. However, if you are [controlling the user-mode debugger from the kernel debugger](#), the maximum line length is 512 characters.

In CDB and KD, press the ENTER key by itself to repeat the previous command. In WinDbg, you can enable or disable this behavior. For more information about this behavior, see [ENTER \(Repeat Last Command\)](#).

If the last command that you issued presents a long display and you want to cut it off, use the [**CTRL+C**](#) key in CDB or KD. In WinDbg, use **Debug | Break** or press CTRL+BREAK.

In kernel-mode debugging, you can cancel commands from the keyboard of the target computer by pressing **CTRL+C**.

You can use the [.cls \(Clear Screen\)](#) command to clear all of the text from the [Debugger Command window](#). This command clears the whole command history. In WinDbg, you can clear the command history by using the **Edit | Clear Command Output** command or by selecting **Clear command output** on the shortcut menu of the Debugger Command window.

Expression Syntax

Many commands and extension commands accept *expressions* as their arguments. The debugger evaluates these expressions before executing the command. For more information about expressions, see [Evaluating Expressions](#).

Aliases

Aliases are text macros that you can use to avoid having to retype complex phrases. There are two kinds of aliases. For more information about aliases, see [Using Aliases](#).

Self-Repeating Commands

You can use the following commands to repeat an action or conditionally execute other commands:

- The [j \(Execute If-Else\)](#) conditional command
- The [z \(Execute While\)](#) conditional command
- The [~e \(Thread-Specific Command\)](#) command qualifier
- The [!list](#) extension command

For more information about each command, see the individual command topics.

Controlling Scrolling

You can use the scrollbar to view your previous commands and their output.

When you are using CDB or KD, any keyboard entry automatically scrolls down the Debugger Command window back to the bottom.

In WinDbg, the display automatically scrolls down to the bottom whenever a command produces output or you press the ENTER key. If you want to disable this automatic scrolling, select the **Options** on the **View** menu and then clear the **Automatically scroll** check box.

WinDbg Text Features

In WinDbg, you can use several additional features to change how text is displayed in the [Debugger Command window](#). You can access some of these features in the WinDbg window, some in the shortcut menu in the Debugger Command window, and some by selecting the appropriate menu icon.

- The **Word wrap** command on the shortcut menu turns on and off the word wrap status. This command affects the whole window, not only commands that you use after this state is changed. Because many commands and extensions produce formatted displays, we typically do not recommend word wrap.
- The **Edit | Add to Command Output** menu command adds a comment in the Debugger Command window. The **Add to command output** command on the shortcut menu has the same effect.
- You can customize the colors that are used for the text and the background of the Debugger Command window. You can specify different colors for different kinds of text. For example, you can display the automatic register output in one color, error messages in another color, and **DbgPrint** messages in a third color.
- You can use all of the features common to WinDbg's debugging information windows, such as customizing the fonts and using special editing commands.

Remote Debugging

When you are performing remote debugging through the debugger, the debugging client can access a limited number of commands. To change the number of commands that the client can access, use the [-clines command-line option](#) or the [_NT_DEBUG_HISTORY_SIZE environment variable](#).

Evaluating Expressions

Article • 10/25/2023

The debugger understands two different forms of expressions: *MASM expressions* and *C++ expressions*.

Microsoft Macro Assembler (MASM) expressions are used in the examples in this Help documentation, except when otherwise noted. In MASM expressions, all symbols are treated as addresses.

C++ expressions are the same as those used in actual C++ code. In these expressions, symbols are understood as the appropriate data types.

When Each Syntax is Used

You can select the default expression evaluator in one of the following ways:

- Use the `_NT_EXPR_EVAL` environment variable before the debugger is started.
- Use the `-ee {masm|c++}` command-line option when the debugger is started.
- Use the `.expr (Choose Expression Evaluator)` command to display or change the expression evaluator after the debugger is running.

If you do not use one of the preceding methods, the debugger uses the MASM expression evaluator.

If you want to evaluate an expression without changing the debugger state, you can use the `? (Evaluate Expression)` command.

All commands and debugging information windows interpret their arguments through the default expression evaluator, with the following exceptions:

- The `?? (Evaluate C++ Expression)` command always uses the C++ expression evaluator.
- The Watch window always uses the C++ expression evaluator.
- The `Locals window` always uses the C++ expression evaluator.
- Some extension commands always use the MASM expression evaluator (and other extension commands accept only numeric arguments instead of full expressions).

- If any part of an expression is enclosed in parentheses and you insert two at signs (@@) before the expression, the expression is evaluated by the expression evaluator that would not typically be used in this case.

The two at signs (@@) enable you to use two different evaluators for different parameters of a single command. It also enables you to evaluate different pieces of a long expression by different methods. You can nest the two at signs. Each appearance of the two at signs switches to the other expression evaluator.

Warning C++ expression syntax is useful for manipulating structures and variables, but it is not well-suited as a parser for the parameters of debugger commands. When you are using debugger commands for general purposes or you are using debugger extensions, you should set MASM expression syntax as the default expression evaluator. If you must have a specific parameter use C++ expression syntax, use the two at sign (@@) syntax.

For more information about the two different expression types, see [Numerical Expression Syntax](#).

Numbers in Expressions

Numbers in MASM expressions are interpreted according to the current radix. The [n \(Set Number Base\)](#) command can be used to set the default radix to 16, 10, or 8. All unprefixed numbers will be interpreted in this base. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Numbers in C++ expressions are interpreted as decimal numbers unless you specify differently. To specify a hexadecimal integer, add **0x** before the number. To specify an octal integer, add **0** (zero) before the number. (However, in the debugger's *output*, the **0n** decimal prefix is sometimes used.)

If you want to display a number in several bases at the same time, use the [.formats \(Show Number Formats\)](#) command.

Symbols in Expressions

The two types of expressions interpret symbols differently:

- In MASM expressions, each symbol is interpreted as an address. Depending on what the symbol refers to, this address is the address of a global variable, local variable, function, segment, module, or any other recognized label.

- In C++ expressions, each symbol is interpreted according to its type. Depending on what the symbol refers to, it might be interpreted as an integer, a data structure, a function pointer, or any other data type. A symbol that does not correspond to a C++ data type (such as an unmodified module name) creates a syntax error.

If a symbol might be ambiguous, precede it with the module name and an exclamation point (!). If the symbol name could be interpreted as a hexadecimal number, precede it with the module name and an exclamation point (!) or only an exclamation point. In order to specify that a symbol is meant to be local, omit the module name, and include a dollar sign and an exclamation point (\$!) before the symbol name. For more information about interpreting symbols, see [Symbol Syntax and Symbol Matching](#).

Operators in Expressions

Each expression type uses a different collection of operators.

For more information about the operators that you can use in MASM expressions and their precedence rules, see [MASM Numbers and Operators](#).

For more information about the operators that you can use in C++ expressions and their precedence rules, see [C++ Numbers and Operators](#).

Remember that MASM operations are always byte-based, and C++ operations follow C++ type rules (including the scaling of pointer arithmetic).

For some examples of the different syntaxes, see [Mixed Expression Examples](#).

Using Shell Commands

Article • 10/25/2023

The debugger can transmit certain commands to the Microsoft Windows environment in which the debugger is running.

You can use the [.shell \(Command Shell\)](#) command in any Windows debugger. With this command, you can execute an application or a Microsoft MS-DOS command directly from the debugger. If you are performing [remote debugging](#), these shell commands are executed on the server.

The [.noshell \(Prohibit Shell Commands\)](#) command or the [-noshell command-line option](#) disables all shell commands. The commands are disabled while the debugger is running, even if you begin a new debugging session. The commands remain disabled even if you issue a [.restart \(Restart Kernel Connection\)](#) command in KD.

If you are running a debugging server, you might want to disable shell commands. If the shell is available, a remote connection can use the [.shell](#) command to change your computer.

Network Drive Control

In WinDbg, you can use the [File | Map Network Drive](#) and [File | Disconnect Network Drive](#) commands to control the network drive mappings. These changes always occur on the computer that WinDbg is running on, never on any computer that is remotely connected to WinDbg.

Using Aliases

Article • 10/25/2023

Aliases are character strings that are automatically replaced with other character strings. You can use them in debugger commands and to avoid retyping certain common phrases.

An alias consists of an *alias name* and an *alias equivalent*. When you use an alias name as part of a debugger command, the name is automatically replaced by the alias equivalent. This replacement occurs immediately, before the command is parsed or executed.

The debugger supports three kinds of aliases:

- You can set and name *user-named aliases*.
- You can set *fixed-name aliases*, but they are named \$u0, \$u1, ..., \$u9.
- The debugger sets and names *automatic aliases*.

Defining a User-Named Alias

When you define a user-named alias, you can choose the alias name and the alias equivalent:

- The alias name can be any string that does not contain white space.
- The alias equivalent can be any string. If you enter it at the keyboard, the alias equivalent cannot contain leading spaces or carriage returns. Alternatively, you can set it equal to a string in memory, the value of a numeric expression, the contents of a file, the value of an environment variable, or the output of one or more debugger commands.

Both the alias name and the alias equivalent are case sensitive.

To define or redefine a user-named alias, use the [as \(Set Alias\)](#) or [aS \(Set Alias\)](#) command.

To remove an alias, use the [ad \(Delete Alias\)](#) command.

To list all current user-named aliases, use the [al \(List Aliases\)](#) command.

Defining a Fixed-Name Alias

There are 10 fixed-name aliases. Their alias names are `$u0`, `$u1`, ..., `$u9`. Their alias equivalents can be any string that does not contain the ENTER keystroke.

Use the [r \(Registers\)](#) command to define the alias equivalents for fixed-name aliases. When you define a fixed-name alias, you must insert a period (.) before the letter "u". The text after the equal sign (=) is the alias equivalent. The alias equivalent can include spaces or semicolons, but leading and trailing spaces are ignored. You should not enclose the alias equivalent in quotation marks (unless you want quotation marks in the results).

Note Do not be confused by using the [r \(Registers\)](#) command for fixed-name aliases. These aliases are not registers or pseudo-registers, even though you use the [r](#) command to set their alias equivalents. You do not have to add an at (@) sign before these aliases, and you cannot use the [r](#) command to *display* the value of one of these aliases.

By default, if you do not define a fixed-name alias, it is an empty string.

Automatic Aliases

The debugger sets the following automatic aliases.

Alias name	Alias equivalent
<code>\$ntnsym</code>	The most appropriate module for NT symbols on the computer's native architecture. This alias can equal either <code>ntdll</code> or <code>nt</code> .
<code>\$ntwsym</code>	The most appropriate module for NT symbols during 32-bit debugging that uses WOW64. This alias could be <code>ntdll32</code> or some other 32-bit version of <code>Ntdll.dll</code> .
<code>\$ntsym</code>	The most appropriate module for NT symbols that match the current machine mode. When you are debugging in native mode, this alias is the same as <code>\$ntnsym</code> . When you are debugging in a non-native mode, the debugger tries to find a module that matches this mode. (For example, during 32-bit debugging that uses WOW64, this alias is the same as <code>\$ntwsym</code> .)
<code>\$CurrentDumpFile</code>	The name of the last dump file that the debugger loaded.
<code>\$CurrentDumpPath</code>	The directory path of the last dump file that the debugger loaded.

Alias name	Alias equivalent
\$CurrentDumpArchiveFile	The name of the last dump archive file (CAB file) that the debugger loaded.
\$CurrentDumpArchivePath	The directory path of the last dump archive file (CAB file) that the debugger loaded.

Automatic aliases are similar to [automatic pseudo-registers](#), except that you can use automatic aliases with alias-related tokens (such as \${ }), while you cannot use pseudo-registers with these tokens.

Using an Alias in the Debugger Command Window

After you define an alias, you can use it in any command entry. The alias name is automatically replaced with the alias equivalent. Therefore, you can use the alias as an expression or as a macro.

An alias name expands correctly even if it is enclosed in quotation marks. Because the alias equivalent can include any number of quotation marks or semicolons, the alias equivalent can represent multiple commands.

A user-named alias is recognized only if its name is separated from other characters by white space. The first character of its alias name must begin the line or follow a space, a semicolon, or a quotation mark. The last character of its alias name must end the line or be followed by a space, a semicolon, or a quotation mark.

Note Any text that you enter into the [Debugger Command window](#) that begins with "as", "aS", "ad", or "al" does not receive alias replacement. This restriction prevents the alias commands from being rendered inoperable. However, this restriction also means that commands that follow ad or al on a line do not have their aliases replaced. If you want aliases to be replaced in a line that begins with one of these strings, add a semicolon before the alias.

However, you can use the \${ } token to expand a user-named alias even when it is next to other text. You can also use this token together with certain switches to prevent an alias from being expanded or to display certain alias-related values. For more information about these situations, see [\\${ } \(Alias Interpreter\)](#).

A fixed-name alias expands correctly from any point within a line, regardless of how it is embedded within the text of the line.

You cannot use commands that are available only in WinDbg ([.open](#), [.write_cmd_hist](#) ([Write Command History](#)), [.lsrcpath](#), and [.lsrcfix](#)) and a few additional commands ([.hh](#),

.cls, .wtitle, .remote, kernel-mode .restart, and user-mode .restart) with aliases.

Using an Alias in a Script File

When you use an alias in a script file, you must take special care to make sure that the alias is expanded at the correct time. Consider the following script:

```
text

.foreach (value {dd 610000 L4})
{
    as /x ${/v:myAlias} value + 1
    .echo value myAlias
}

ad myAlias
```

The first time through the loop, the [as, aS \(Set Alias\)](#) command assigns a value to the myAlias. The value assigned to myAlias is 1 plus 610000 (the first output of the dd command). However, when the [.echo \(Echo Comment\)](#) command is executed, myAlias has not yet been expanded, so instead of seeing 610001, we see the text "myAlias".

```
dbgcmd

0:001> $$>< c:\Script02.txt
00610000 myAlias
00905a4d 0x610001
00000003 0x905a4e
00000004 0x4
000fffff 0x5
```

The problem is that myAlias is not expanded until a new block of code is entered. The next entry to the loop is a new block, so myAlias gets expanded to 610001. But it is too late: we should have seen 610001 the first time through the loop, not the second time. We can fix this problem by enclosing the [.echo \(Echo Comment\)](#) command in a new block as shown in the following script.

```
text

.foreach (value {dd 610000 L4})
{
    as /x ${/v:myAlias} value + 1
    .block{.echo value myAlias}
}

ad myAlias
```

With the altered script, we get the following correct output.

```
dbgcmd  
0:001> $$>< c:\Script01.txt  
00610000 0x610001  
00905a4d 0x905a4e  
00000003 0x4  
00000004 0x5  
0000ffff 0x10000
```

For more information, see [.block](#) and [\\${ } \(Alias Interpreter\)](#).

Using a .foreach Token in an Alias

When you use a [.foreach](#) token in the definition of an alias, you must take special care to ensure that the token is expanded. Consider the following sequence of commands.

```
dbgcmd  
r $t0 = 5  
ad myAlias  
.foreach /pS 2 /ps 2 (Token {@$t0}) {as myAlias Token}  
al
```

The first command sets the value of the `$t0` pseudo register to 5. The second command deletes any value that might have been previously assigned to `myAlias`. The third command takes the third token of the `?@$t0` command and attempts to assign the value of that token to `myAlias`. The fourth command lists all aliases and their values. We would expect the value of `myAlias` to be 5, but instead the value is the word "Token".

```
dbgcmd  
-----  
 Alias      Value  
-----  
 myAlias    Token
```

The problem is that the `as` command is at the beginning of the line in the body of the [.foreach](#) loop. When a line begins with an `as` command, aliases and tokens in that line are not expanded. If we put a semicolon or blank space before the `as` command, then any alias or token that already has a value is expanded. In this example, `myAlias` is not expanded because it does not already have a value. `Token` is expanded because it has a value of 5. Here is the same sequence of commands with the addition of a semicolon before the `as` command.

```
dbgcmd
```

```
r $t0 = 5
ad myAlias
.foreach /ps 2 /ps 2 (Token {@$t0}) {;as myAlias Token}
al
```

Now we get the expected output.

```
dbgcmd
```

Alias	Value
-----	-----
myAlias	5

Recursive Aliases

You can use a fixed-name alias in the definition of any alias. You can also use a user-named alias in the definition of a fixed-name alias. However, to use a user-named alias in the definition of another user-named alias, you have to add a semicolon before the `as` or `aS` command, or else the alias replacement does not occur on that line.

When you are using recursive definitions of this type, each alias is translated as soon as it is used. For example, the following example displays 3, not 7.

```
dbgcmd
```

```
0:000> r $.u2=2
0:000> r $.u1=1+$u2
0:000> r $.u2=6
0:000> ? $u1
Evaluate expression: 3 = 00000003
```

Similarly, the following example displays 3, not 7.

```
dbgcmd
```

```
0:000> as fred 2
0:000> r $.u1= 1 + fred
0:000> as fred 6
0:000> ? $u1
Evaluate expression: 3 = 00000003
```

The following example is also permitted and displays 9.

```
dbgcmd
```

```
0:000> r $.u0=2
0:000> r $.u0=7+$u0
0:000> ? $u0
Evaluate expression: 9 = 00000009
```

Examples

You can use aliases so that you do not have to type long or complex symbol names, as in the following example.

```
dbgcmd
```

```
0:000> as Short usersrv!NameTooLongToWantToType
0:000> dw Short +8
```

The following example is similar to the preceding example but it uses a fixed-name alias.

```
dbgcmd
```

```
0:000> r $.u0=usersrv!NameTooLongToWantToType
0:000> dw $u0+8
```

You can use aliases as macros for commands that you use frequently. The following example increments the **eax** and **ebx** registers two times.

```
dbgcmd
```

```
0:000> as GoUp r eax=eax+1; r ebx=ebx+1
0:000> GoUp
0:000> GoUp
```

The following example uses an alias to simplify typing of commands.

```
dbgcmd
```

```
0:000> as Cmd "dd esp 14; g"
0:000> bp MyApi Cmd
```

The following example is similar to the preceding example but it uses a fixed-name alias.

```
dbgcmd
```

```
0:000> r $.u5="dd esp 14; g"
0:000> bp MyApi $u5
```

Both of the preceding examples are equivalent to the following command.

```
dbgcmd
0:000> bp MyApi "dd esp 14; g"
```

Tools.ini File

In CDB (and NTSD), you can predefined fixed-name aliases in the [tools.ini](#) file. To predefined a fixed-name alias, add the \$u fields that you want to your [NTSD] entry, as in the following example.

```
ini
[NTSD]
$u1:_ntdll!_RtlRaiseException
$u2:"dd esp 14;g"
$u9:$u1 + 42
```

You cannot set user-named aliases in the Tools.ini file.

Fixed-Name Aliases vs. User-Named Aliases

User-name aliases are easier to use than fixed-named aliases. Their definition syntax is simpler, and you can list them by using the [al \(List Aliases\)](#) command.

Fixed-named aliases are replaced if they are used next to other text. To make a user-named alias be replaced when it is next to other text, enclose it in the [\\${ } \(Alias Interpreter\)](#) token.

Fixed-name alias replacement occurs before user-named alias replacement.

Using Script Files

Article • 10/25/2023

A *script file* is a text file that contains a sequence of debugger commands. There are a variety of ways for the debugger to load a script file and execute it. A script file can contain commands to be executed sequentially or can use a more complex flow of execution.

To execute a script file, you can do one of the following:

- (KD and CDB only; only when the debugger starts) Create a script file that is named Ntsd.ini and put it in the directory where you are starting the debugger from. The debugger automatically executes this file when the debugger starts. To use a different file for the startup script file, specify the path and file name by using the **-cf command-line option** or by using the **IniFile** entry in the **Tools.ini** file.
- (KD and CDB only; when each session starts) Create a script file and specify its path and file name by using the **-cfr command-line option**. The debugger automatically executes this script file when the debugger starts and every time that the target is restarted.
- Use the \$<, \$><, \$\$<, and \$\$>< commands to execute a script file after the debugger is running. For more information about the syntax, see [\\$<, \\$><, \\$><, \\$\\$>< \(Run Script File\)](#).

The \$>< and \$\$>< commands differ from the other methods of running scripts in one important way. When you use these commands, the debugger opens the specified script file, replaces all carriage returns with semicolons, and executes the resulting text as a single command block. These commands are useful for running scripts that contain debugger command programs. For more information about these programs, see [Using Debugger Command Programs](#).

You cannot use commands that are available only in WinDbg (such as [.Isrfix \(Use Local Source Server\)](#), [.Isrcpath \(Set Local Source Path\)](#), [.open \(Open Source File\)](#), and [.write_cmd_hist \(Write Command History\)](#)) in script files, even if the script file is executed in WinDbg. In addition, you cannot use the [.beep \(Speaker Beep\)](#), [.cls \(Clear Screen\)](#), [.hh \(Open HTML Help File\)](#), [.idle_cmd \(Set Idle Command\)](#), [.remote \(Create Remote.exe Server\)](#), kernel-mode [.restart \(Restart Kernel Connection\)](#), user-mode [.restart \(Restart Target Application\)](#), or [.wtitle \(Set Window Title\)](#) commands in a script file.

WinDbg supports the same scripts as KD and CDB, with one minor exception. You can use the [.remote_exit \(Exit Debugging Client\)](#) command only in a script file that KD or CDB uses. You cannot exit from a debugging client though a script that is executed in WinDbg.

Using Debugger Extensions

Article • 10/25/2023

WinDbg, KD, and CDB all allow the use of debugger extension commands. These extensions give these three Microsoft debuggers a great degree of power and flexibility.

Debugger extension commands are used much like the standard debugger commands. However, while the built-in debugger commands are part of the debugger binaries themselves, debugger extension commands are exposed by DLLs distinct from the debugger.

This allows you to write new debugger commands which are tailored to your specific need. In addition, a number of debugger extension DLLs are shipped with the debugging tools themselves.

This section includes:

[Loading Debugger Extension DLLs](#)

[Using Debugger Extension Commands](#)

[Writing New Debugger Extensions](#)

Loading Debugger Extension DLLs

Article • 10/25/2023

There are several methods of loading debugger extension DLLs, as well as controlling the default debugger extension DLL and the default debugger extension path:

- (Before starting the debugger) Use the `_NT_DEBUGGER_EXTENSION_PATH` environment variable to set the default path for extension DLLs. This can be a number of directory paths, separated by semicolons.
- Use the [.load \(Load Extension DLL\)](#) command to load a new DLL.
- Use the [.unload \(Unload Extension DLL\)](#) command to unload a DLL.
- Use the [.unloadall \(Unload All Extension DLLs\)](#) command to unload all debugger extensions.
- (Before starting the debugger; CDB only) Use the `tools.ini` file to set the default extension DLL.
- (Before starting the debugger) Use the `-a` command-line option to set the default extension DLL.
- Use the [.extpath \(Set Extension Path\)](#) command to set the extension DLL search path.
- Use the [.setdll \(Set Default Extension DLL\)](#) command to set the default extension DLL.
- Use the [.chain \(List Debugger Extensions\)](#) command to display all loaded debugger extension modules, in their default search order.

You can also load an extension DLL simply by using the full `!module.extension` syntax the first time you issue a command from that module. See [Using Debugger Extension Commands](#) for details.

The extension DLLs that you are using must match the operating system of the target computer. The extension DLLs that ship with the Debugging Tools for Windows package are each placed in a different subdirectory of the installation directory:

- The `winxp` directory contains extensions that can be used with Windows XP and later versions of Windows.

- The winext directory contains extensions that can be used with any version of Windows. The dbghelp.dll module, located in the base directory of Debugging Tools for Windows, also contains extensions of this type.

If you write your own debugger extensions, you can place them in any directory. However, it is advised that you place them in a new directory and add that directory to the debugger extension path.

There can be as many as 32 extension DLLs loaded.

Using Debugger Extension Commands

Article • 10/25/2023

The use of debugger extension commands is very similar to the use of [debugger commands](#). The command is typed in the Debugger Command window, producing either output in this window or a change in the target application or target computer.

An actual debugger extension command is an entry point in a DLL called by the debugger.

Debugger extensions are invoked by the following syntax:

![module.]extension [arguments]

The module name should not be followed with the .dll file name extension. If *module* includes a full path, the default string size limit is 255 characters.

If the module has not already been loaded, it will be loaded into the debugger using a call to **LoadLibrary(module)**. After the debugger has loaded the extension library, it calls the **GetProcAddress** function to locate the extension name in the extension module. The extension name is case-sensitive and must be entered exactly as it appears in the extension module's .def file. If the extension address is found, the extension is called.

Search Order

If the module name is not specified, the debugger will search the loaded extension modules for this export.

The default search order is as follows:

1. The extension modules that work with all operating systems and in both modes:
Dbghelp.dll and winext\ext.dll.
2. The extension module that works in all modes but is operating-system-specific. For Windows XP and later versions of Windows, this is winxp\exts.dll.
3. The extension module that works with all operating systems but is mode-specific.
For kernel mode, this is winext\kext.dll. For user mode, this is winext\uext.dll.
4. The extension module that is both operating-system-specific and mode-specific.
The following table specifies this module.

User Mode	Kernel Mode
winxp \ ntsdexts.dll	winxp \ kdexts.dll

When an extension module is unloaded, it is removed from the search chain. When an extension module is loaded, it is added to the beginning of the search order. The [.setdll \(Set Default Extension DLL\)](#) command can be used to promote any module to the top of the search chain. By using this command repeatedly, you can completely control the search chain.

Use the [.chain \(List Debugger Extensions\)](#) command to display a list of all loaded extension modules in their current search order.

If you attempt to execute an extension command that is not in any of the loaded extension modules, you will get an Export Not Found error message.

Writing New Debugger Extensions

Article • 10/25/2023

You can create your own debugging commands by writing an extension DLL. For example, you might want to write a command to display a complex data structure, or a command that will stop and start the target depending on the value of certain variables or memory locations.

There are two different types of debugger extensions:

- *DbgEng extensions*. These are based on the prototypes in the dbgeng.h header file, and also those in the wdbgexts.h header file.
- *WdbgExts extensions*. These are based on the prototypes in the wdbgexts.h header file alone.

For information about how to write debugger extensions, see [Writing DbgEng Extensions](#) and [Writing WdbgExts Extensions](#).

Debugger Commands

Article • 10/25/2023

This section includes the following topics:

[Syntax Rules](#)

[Command Tokens](#)

[Commands](#)

[Meta-Commands](#)

[Control Keys](#)

Syntax Rules

Article • 10/25/2023

This section describes the syntax rules that you must follow to use debugger commands.

When you are debugging, you should obey the following general syntax rules:

- You can use any combination of uppercase and lowercase letters in commands and arguments, except when specifically noted in the topics in this section.
- You can separate multiple command parameters by one or more spaces or by a comma (,).
- You can typically omit the space between a command and its first parameter . You can frequently omit other spaces if this omission does not cause any ambiguity.

The command reference topics in this section use the following items:

- Characters in **bold** font style indicate items that you must literally type.
- Characters in *italic* font style indicate parameters that are explained in the "Parameters" section of the reference topic.
- Parameters in brackets ([xxx]) are optional. Brackets with a vertical bar ([xxx|yyy]) indicate that you can use one, or none, of the enclosed parameters.
- Braces with vertical bars ({xxx|yyy}) indicate that you must use exactly one of the enclosed parameters .

The following topics describe the syntax that the following parameter types use:

[Numerical Expression Syntax](#)

[String Wildcard Syntax](#)

[Register Syntax](#)

[Pseudo-Register Syntax](#)

[Source Line Syntax](#)

[Address and Address Range Syntax](#)

[Thread Syntax](#)

[Process Syntax](#)

[System Syntax](#)

[Multiprocessor Syntax](#)

Syntax also plays an important role in using symbols. For further details, see [Symbol Syntax and Symbol Matching](#).

Numerical Expression Syntax

Article • 10/25/2023

The debugger accepts two different kinds of numeric expressions: *C++ expressions* and *MASM expressions*. Each of these expressions follows its own syntax rules for input and output.

For more information about when each syntax type is used, see [Evaluating Expressions](#).

Use the [.expr \(Choose Expression Evaluator\)](#) command to display or change the expression evaluator after the debugger is running.

This section includes the following topics:

[MASM Numbers and Operators](#)

[C++ Numbers and Operators](#)

[MASM Expressions vs. C++ Expressions](#)

[Mixed Expression Examples](#)

[Sign Extension](#)

[? \(Evaluate Expression\)](#)

[.expr \(Choose Expression Evaluator\)](#)

MASM Numbers and Operators

Article • 10/25/2023

This topic describes the use of Microsoft Macro Assembler (MASM) expression syntax with the Windows Debugging tools.

The debugger accepts two different kinds of numeric expressions: C++ expressions and MASM expressions. Each of these expressions follows its own syntax rules for input and output.

For more information about when each syntax type is used, see [Evaluating Expressions](#) and [? \(Evaluate Expression\)](#).

In this example the ? command displays the value of the instruction pointer register using the MASM expression evaluator.

```
dbgcmd  
0:000> ? @rip  
Evaluate expression: 140709230544752 = 00007ff9`6bb40770
```

Set the Expression Evaluator to MASM

Use the [.expr \(Choose Expression Evaluator\)](#) to see what the default expression evaluator is and change it to MASM.

```
dbgcmd  
0:000> .expr /s masm  
Current expression evaluator: MASM - Microsoft Assembler expressions
```

Now that the default expression evaluator has been changed, the [? \(Evaluate Expression\)](#) command can be used to display MASM expressions. This example adds the hex value of 8 to the rip register.

```
dbgcmd  
0:000> ? @rip + 8  
Evaluate expression: 140709230544760 = 00007ff9`6bb40778
```

The register reference of @rip is described in more detail in [Register Syntax](#).

Numbers in Debugger MASM Expressions

You can put numbers in MASM expressions in base 16, 10, 8, or 2.

Use the [n \(Set Number Base\)](#) command to set the default radix to 16, 10, or 8. All unprefixed numbers are then interpreted in this base. You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

You can also specify hexadecimal numbers by adding an **h** after the number. You can use uppercase or lowercase letters within numbers. For example, "0x4AB3", "0X4aB3", "4AB3h", "4ab3h", and "4aB3H" have the same meaning.

If you do not add a number after the prefix in an expression, the number is read as 0. Therefore, you can write 0 as 0, the prefix followed by 0, and only the prefix. For example, in hexadecimal, "0", "0x0", and "0x" have the same meaning.

You can enter hexadecimal 64-bit values in the **xxxxxxxx`xxxxxxxx** format. You can also omit the grave accent (`). If you include the grave accent, [automatic sign extension](#) is disabled.

This example shows how to add a decimal, octal and binary value to register 10.

```
dbgcmd  
? @r10 + 0x10 + 0t10 + 0y10  
Evaluate expression: 26 = 00000000`0000001a
```

Symbols in Debugger MASM Expressions

In MASM expressions, the numeric value of any symbol is its memory address. Depending on what the symbol refers to, this address is the address of a global variable, local variable, function, segment, module, or any other recognized label.

To specify which module the address is associated with, include the module name and an exclamation point (!) before the name of the symbol. If the symbol could be interpreted as a hexadecimal number, include the module name and an exclamation point, or just an exclamation point, before the symbol name. For more information about symbol recognition, see [Symbol Syntax and Symbol Matching](#).

Use two colons (::) or two underscores (_) to indicate the members of a class.

Use a grave accent (`) or an apostrophe (') in a symbol name only if you add a module name and exclamation point before the symbol.

Numeric Operators in MASM Expressions

You can modify any component of an expression by using a unary operator. You can combine any two components by using a binary operator. Unary operators take precedence over binary operators. When you use multiple binary operators, the operators follow the fixed precedence rules that are described in the following tables.

You can always use parentheses to override precedence rules.

If part of an MASM expression is enclosed in parentheses and two at signs (@@) appear before the expression, the expression is interpreted according to [C++ expression rules](#). You cannot add a space between the two at signs and the opening parenthesis. You can also specify the [expression evaluator](#) by using @@c++(...) or @@masm(...).

When you perform arithmetic computations, the MASM expression evaluator treats all numbers and symbols as ULONG64 types.

Unary address operators assume DS as the default segment for addresses. Expressions are evaluated in order of operator precedence. If adjacent operators have equal precedence, the expression is evaluated from left to right.

You can use the following unary operators.

Operator	Meaning
+	Unary plus
-	Unary minus
not	Returns 1 if the argument is zero. Returns zero for any nonzero argument.
hi	High 16 bits
low	Low 16 bits
by	Low-order byte from the specified address.
\$pby	Same as by except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
wo	Low-order word from the specified address.

Operator	Meaning
\$pwo	Same as wo except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
dwo	Double-word from the specified address.
\$pdwo	Same as dwo except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
qwo	Quad-word from the specified address.
\$pqwo	Same as qwo except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.
poi	Pointer-sized data from the specified address. The pointer size is 32 bits or 64 bits. In kernel debugging, this size is based on the processor of the <i>target</i> computer. Therefore, poi is the best operator to use if you want pointer-sized data.
\$ppoi	Same as poi except that it takes a physical address. Only physical memory that uses the default caching behavior can be read.

Examples

The following example shows how to use **poi** to dereference a pointer to see the value that is stored at that memory location.

First determine the memory address of interest. For example we can look at the thread structure and decide we want to see the value of the CurrentLocale.

```
dbgcmd

0:000> dx @$teb
@$teb          : 0x8eed57b000 [Type: _TEB *]
  [+0x000] NtTib          [Type: _NT_TIB]
  [+0x038] EnvironmentPointer : 0x0 [Type: void *]
  [+0x040] ClientId        [Type: _CLIENT_ID]
  [+0x050] ActiveRpcHandle  : 0x0 [Type: void *]
  [+0x058] ThreadLocalStoragePointer : 0x1f8f9d634a0 [Type: void *]
  [+0x060] ProcessEnvironmentBlock : 0x8eed57a000 [Type: _PEB *]
  [+0x068] LastErrorValue   : 0x0 [Type: unsigned long]
  [+0x06c] CountOfOwnedCriticalSection : 0x0 [Type: unsigned long]
```

```
[+0x070] CsrClientThread : 0x0 [Type: void *]  
[+0x078] Win32ThreadInfo : 0x0 [Type: void *]  
[+0x080] User32Reserved [Type: unsigned long [26]]  
[+0x0e8] UserReserved : [Type: unsigned long [5]]  
[+0x100] WOW32Reserved : 0x0 [Type: void *]  
[+0x108] CurrentLocale : 0x409 [Type: unsigned long]
```

CurrentLocale is located 0x108 beyond the start of the TEB.

```
dbgcmd  
  
0:000> ? @$teb + 0x108  
Evaluate expression: 613867303176 = 0000008e`ed57b108
```

Use poi to dereference that address.

```
dbgcmd  
  
0:000> ? poi(0000008e`ed57b108)  
Evaluate expression: 1033 = 00000000`00000409
```

The returned value of 409 matches value of CurrentLocale in the TEB structure.

Or use poi and parentheses to dereference the calculated address.

```
dbgcmd  
  
0:000> ? poi(@$teb + 0x108)  
Evaluate expression: 1033 = 00000000`00000409
```

Use the *by* or *wo* unary operators to return a byte or word from the target address.

```
dbgcmd  
  
0:000> ? by(0000008e`ed57b108)  
Evaluate expression: 9 = 00000000`00000009  
0:000> ? wo(0000008e`ed57b108)  
Evaluate expression: 1033 = 00000000`00000409
```

Binary Operators

You can use the following binary operators. The operators in each cell take precedence over those in lower cells. Operators in the same cell are of the same precedence and are parsed from left to right.

Operator	Meaning
*	Multiplication
/	Integer division
mod (or %)	Modulus (remainder)
+	Addition
-	Subtraction
<<	Left shift
>>	Logical right shift
>>>	Arithmetic right shift
= (or ==)	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
and (or &)	Bitwise AND
xor (or ^)	Bitwise XOR (exclusive OR)
or (or)	Bitwise OR

The <, >, =, ==, and != comparison operators evaluate to 1 if the expression is true or zero if the expression is false. A single equal sign (=) is the same as a double equal sign (==). You cannot use side effects or assignments within a MASM expression.

An invalid operation (such as division by zero) results in an "Operand error" is returned to the [Debugger Command window](#).

We can check that the returned value matches 0x409 by using the == comparison operator.

```
dbgcmd
```

```
0:000> ? poi(@$teb + 0x108)==0x409
Evaluate expression: 1 = 00000000`00000001
```

Non-Numeric Operators in MASM Expressions

You can also use the following additional operators in MASM expressions.

Operator	Meaning
<code>\$fnsucc(FnAddress, RetVal, Flag)</code>	Interprets the <i>RetVal</i> value as a return value for the function that is located at the <i>FnAddress</i> address. If this return value qualifies as a success code, <code>\$fnsucc</code> returns TRUE . Otherwise, <code>\$fnsucc</code> returns FALSE . If the return type is BOOL , bool , HANDLE , HRESULT , or NTSTATUS , <code>\$fnsucc</code> correctly understands whether the specified return value qualifies as a success code. If the return type is a pointer, all values other than NULL qualify as success codes. For any other type, success is defined by the value of <i>Flag</i> . If <i>Flag</i> is 0, a nonzero value of <i>RetVal</i> is success. If <i>Flag</i> is 1, a zero value of <i>RetVal</i> is success.
<code>\$iment (Address)</code>	Returns the address of the image entry point in the loaded module list. <i>Address</i> specifies the Portable Executable (PE) image base address. The entry is found by looking up the image entry point in the PE image header of the image that <i>Address</i> specifies. You can use this function for both modules that are already in the module list and to set unresolved breakpoints by using the bu command.
<code>\$scmp("String1", "String2")</code>	Evaluates to -1, 0, or 1, like the strcmp by using the strcmp C function .
<code>\$sicmp("String1", "String2")</code>	Evaluates to -1, 0, or 1, like the stricmp Microsoft Win32 function .
<code>\$spat("String", "Pattern")</code>	Evaluates to TRUE or FALSE depending on whether <i>String</i> matches <i>Pattern</i> . The matching is case-insensitive. <i>Pattern</i> can contain a variety of wildcard characters and specifiers. For more information about the syntax, see String Wildcard Syntax .
<code>\$vvalid(Address, Length)</code>	Determines whether the memory range that begins at <i>Address</i> and extends for <i>Length</i> bytes is valid. If the memory is valid, <code>\$vvalid</code> evaluates

Operator	Meaning
	to 1. If the memory is invalid, \$vvalid evaluates to 0.

Examples

The following shows how to use the investigate the range of valid memory around a loaded module

First determine the address of the area of interest, for example by using the [!m \(List Loaded Modules\)](#) command.

```
dbgcmd

0:000> !m
start           end           module name
00007ff6`0f620000 00007ff6`0f658000  notepad    (deferred)
00007ff9`591d0000 00007ff9`5946a000  COMCTL32  (deferred)
...
```

Use \$vvalid to check a memory range.

```
dbgcmd

0:000> ? $vvalid(0x00007ff60f620000, 0xFFFF)
Evaluate expression: 1 = 00000000`00000001
```

Use \$vvalid to confirm that this larger range, is an invalid memory range.

```
dbgcmd

0:000> ? $vvalid(0x00007ff60f620000, 0xFFFF)
Evaluate expression: 0 = 00000000`00000000
```

This is also an invalid range.

```
dbgcmd

0:000> ? $vvalid(0x0, 0xF)
Evaluate expression: 0 = 00000000`00000000
```

Use *not* to return zero when the memory range is valid.

```
dbgcmd
```

```
0:000> ? not($vvalid(0x00007ff60f620000, 0xFFFF))  
Evaluate expression: 0 = 00000000`00000000
```

Use \$imnet to look at the entry point of COMCTL32 that we previously used the `!m` command to determine the address. It starts at `00007ff9`591d0000`.

```
dbgcmd
```

```
0:000> ? $iment(00007ff9`591d0000)  
Evaluate expression: 140708919287424 = 00007ff9`59269e80
```

Disassemble the returned address to examine the entry point code.

```
dbgcmd
```

```
0:000> u 00007ff9`59269e80  
COMCTL32!DllMainCRTStartup:  
00007ff9`59269e80 48895c2408      mov     qword ptr [rsp+8],rbx  
00007ff9`59269e85 4889742410      mov     qword ptr [rsp+10h],rsi  
00007ff9`59269e8a 57                push    rdi
```

COMCTL32 is displayed in the output confirming this is the entry point for this module.

Registers and Pseudo-Registers in MASM Expressions

You can use registers and pseudo-registers within MASM expressions. You can add an at sign (@) before all registers and pseudo-registers. The at sign causes the debugger to access the value more quickly. This @ sign is unnecessary for the most common x86-based registers. For other registers and pseudo-registers, we recommend that you add the at sign, but it is not actually required. If you omit the at sign for the less common registers, the debugger tries to parse the text as a hexadecimal number, then as a symbol, and finally as a register.

You can also use a period (.) to indicate the current instruction pointer. You should not add an @ sign before this period, and you cannot use a period as the first parameter of the [r command](#). This period has the same meaning as the `$ip` pseudo-register.

For more information about registers and pseudo-registers, see [Register Syntax](#) and [Pseudo-Register Syntax](#).

Use the r register command to see that the value of the @rip register is 00007ffb`7ed00770.

```
dbgcmd

0:000> r
rax=0000000000000000 rbx=0000000000000010 rcx=00007ffb7eccd2c4
rdx=0000000000000000 rsi=00007ffb7ed61a80 rdi=00000027eb6a7000
rip=00007ffb7ed00770 rsp=00000027eb87f320 rbp=0000000000000000
r8=00000027eb87f318 r9=0000000000000000 r10=0000000000000000
r11=0000000000000246 r12=000000000000040 r13=0000000000000000
r14=00007ffb7ed548f0 r15=00000210ea090000
iopl=0      nv up ei pl zr na po nc
cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b
efl=00000246
ntdll!LdrpDoDebuggerBreak+0x30:
00007ffb`7ed00770 cc          int     3
```

This same value can be displayed using the . period shortcut.

```
dbgcmd

0:000> ?
Evaluate expression: 140718141081456 = 00007ffb`7ed00770
```

We can confirm that those values are all equivalent, and return zero if they are, using this MASM expression.

```
dbgcmd

0:000> ? NOT(($ip = .) AND ($ip = @rip) AND (@rip =. ))
Evaluate expression: 0 = 00000000`00000000
```

Source Line Numbers in MASM Expressions

You can use source file and line number expressions within MASM expressions. You must enclose these expressions by using grave accents (`). For more information about the syntax, see [Source Line Syntax](#).

See also

[MASM Expressions vs. C++ Expressions](#)

[Mixed Expression Examples](#)

C++ Numbers and Operators

Sign Extension

? (Evaluate Expression)

C++ numbers and operators

Article • 10/25/2023

This article describes the use of C++ expression syntax with the Windows debugging tools.

The debugger accepts two different kinds of numeric expressions: C++ expressions and Microsoft Macro Assembler (MASM) expressions. Each of these expressions follows its own syntax rules for input and output.

For more information about when each syntax type is used, see [Evaluating expressions](#) and the [? evaluate expression](#) command.

The C++ expression parser supports all forms of C++ expression syntax. The syntax includes all data types, including pointers, floating-point numbers, and arrays, and all C++ unary and binary operators.

The *Watch* and the *Locals* windows in the debugger always use the C++ expression evaluator.

In the following example, the [?? evaluate C++ expression](#) command displays the value of the instruction pointer register.

```
dbgcmd  
0:000> ?? @eip  
unsigned int 0x771e1a02
```

We can use the C++ `sizeof` function to determine the size of structures.

```
dbgcmd  
0:000> ?? (sizeof(_TEB))  
unsigned int 0x1000
```

Set the expression evaluator to C++

Use the [.expr choose expression](#) evaluator to see the default expression evaluator and change it to C++.

```
dbgcmd
```

```
0:000> .expr
Current expression evaluator: MASM - Microsoft Assembler expressions
0:000> .expr /s c++
Current expression evaluator: C++ - C++ source expressions
```

After the default expression evaluator has been changed, the [? evaluate expression](#) command can be used to display C++ expressions. The following example displays the value of the instruction pointer register.

```
dbgcmd

0:000> ? @eip
Evaluate expression: 1998461442 = 771e1a02
```

To learn more about the `@eip` register reference, see [Register syntax](#).

In this example, the hex value of 0xD is added to the eip register.

```
dbgcmd

0:000> ? @eip + 0xD
Evaluate expression: 1998461455 = 771e1a0f
```

Registers and pseudo-registers in C++ expressions

You can use registers and pseudo-registers within C++ expressions. The @ sign must be added before the register or pseudo-register.

The expression evaluator automatically performs the proper cast. Actual registers and integer-value pseudo-registers are cast to `ULONG64`. All addresses are cast to `PUCHAR`, `$thread` is cast to `ETHREAD*`, `$proc` is cast to `EPROCESS*`, `$teb` is cast to `TEB*`, and `$peb` is cast to `PEB*`.

This example displays the TEB.

```
dbgcmd

0:000> ?? @$teb
struct _TEB * 0x004ec000
+0x000 NtTib          : _NT_TIB
+0x01c EnvironmentPointer : (null)
+0x020 ClientId       : _CLIENT_ID
```

```
+0x028 ActiveRpcHandle : (null)
+0x02c ThreadLocalStoragePointer : 0x004ec02c Void
+0x030 ProcessEnvironmentBlock : 0x004e9000 _PEB
+0x034 LastErrorValue : 0xbb
+0x038 CountOfOwnedCriticalSection : 0
```

You can't change a register or pseudo-register by an assignment or side-effect operator. You must use the [r registers](#) command to change these values.

The following example sets the pseudo register to a value of 5 and then displays it.

```
dbgcmd

0:000> r $t0 = 5

0:000> ?? @$t0
unsigned int64 5
```

For more information about registers and pseudo-registers, see [Register syntax](#) and [Pseudo-register syntax](#).

Numbers in C++ expressions

Numbers in C++ expressions are interpreted as decimal numbers, unless you specify them in another manner. To specify a hexadecimal integer, add 0x before the number. To specify an octal integer, add 0 (zero) before the number.

The default debugger radix doesn't affect how you enter C++ expressions. You can't directly enter a binary number, except by nesting a MASM expression within the C++ expression.

You can enter a hexadecimal 64-bit value in the xxxxxxxx`xxxxxxxx format. You can also omit the grave accent (`). Both formats produce the same value.

You can use the `L`, `u`, and `I64` suffixes with integer values. The actual size of the number that's created depends on the suffix and the number that you enter. For more information about this interpretation, see a C++ language reference.

The *output* of the C++ expression evaluator keeps the data type that the C++ expression rules specify. However, if you use this expression as an argument for a command, a cast is always made. For example, you don't have to cast integer values to pointers when they're used as addresses in command arguments. If the expression's value can't be validly cast to an integer or a pointer, a syntax error occurs.

You can use the `0n` (decimal) prefix for some *output*, but you can't use it for C++ expression input.

Characters and strings in C++ expressions

You can enter a character by surrounding it with single quotation marks ('). The standard C++ escape characters are permitted.

You can enter string literals by surrounding them with double quotation marks (""). You can use \" as an escape sequence within such a string. However, strings have no meaning to the [expression evaluator](#).

Symbols in C++ expressions

In a C++ expression, each symbol is interpreted according to its type. Depending on what the symbol refers to, it might be interpreted as an integer, a data structure, a function pointer, or any other data type. A syntax error occurs if you use a symbol that doesn't correspond to a C++ data type, such as an unmodified module name, within a C++ expression.

You can use a grave accent (`) or an apostrophe (') in a symbol name only if you add a module name and exclamation point before the symbol name. When you add the < and > delimiters after a template name, you can add spaces between these delimiters.

If the symbol might be ambiguous, you can add a module name and an exclamation point (!) or only an exclamation point before the symbol. To specify that a symbol is meant to be local, omit the module name, and include a dollar sign and an exclamation point (\$!) before the symbol name. For more information about symbol recognition, see [Symbol syntax and symbol matching](#).

Structures in C++ expressions

The C++ expression evaluator casts pseudo-registers to their appropriate types. For example, `$teb` is cast as a `TEB*`.

```
dbgcmd

0:000> ?? @$teb
struct _TEB * 0x004ec000
+0x000 NtTib          : _NT_TIB
+0x01c EnvironmentPointer : (null)
+0x020 ClientId       : _CLIENT_ID
```

```
+0x028 ActiveRpcHandle : (null)
+0x02c ThreadLocalStoragePointer : 0x004ec02c Void
+0x030 ProcessEnvironmentBlock : 0x004e9000 _PEB
+0x034 LastErrorValue : 0xbb
+0x038 CountOfOwnedCriticalSection : 0
```

The following example displays the process ID in the TEB structure showing the use of a pointer to a member of referenced structure.

```
dbgcmd

0:000> ?? @$teb->ClientId.UniqueProcess
void * 0x0000059c
```

Operators in C++ expressions

You can use parentheses to override precedence rules.

If you enclose part of a C++ expression in parentheses and add two *at* signs (@@) before the expression, the expression is interpreted according to MASM expression rules. You can't add a space between the two *at* signs and the opening parenthesis. The final value of this expression is passed to the C++ expression evaluator as a ULONG64 value. You can also specify the expression evaluator by using @@c++(...) or @@masm(...).

Data types are indicated as usual in the C++ language. The symbols that indicate arrays ([]), pointer members (->), UDT members (.), and members of classes (::) are all recognized. All arithmetic operators are supported, including assignment and side-effect operators. However, you can't use the `new`, `delete`, and `throw` operators, and you can't actually call a function.

Pointer arithmetic is supported and offsets are scaled correctly. Note that you can't add an offset to a function pointer. If you must add an offset to a function pointer, cast the offset to a character pointer first.

As in C++, if you use operators with invalid data types, a syntax error occurs. The debugger's C++ expression parser uses slightly more relaxed rules than most C++ compilers, but all major rules are enforced. For example, you can't shift a non-integer value.

You can use the following operators. The operators in each cell take precedence over operators in lower cells. Operators in the same cell are of the same precedence and are parsed from left to right.

As with C++, expression evaluation ends when its value is known. This ending enables you to effectively use expressions such as `?? myPtr && *myPtr`.

Reference and type casting

Operator	Meaning
<i>Expression // Comment</i>	Ignore all subsequent text
<i>Class :: Member</i>	Member of class
<i>Class ::~Member</i>	Member of class (destructor)
<code>:: Name</code>	Global
<i>Structure . Field</i>	Field in a structure
<i>Pointer -> Field</i>	Field in referenced structure
<i>Name [integer]</i>	Array subscript
<i>LValue ++</i>	Increment (after evaluation)
<i>LValue --</i>	Decrement (after evaluation)
<code>dynamic_cast <type>(Value)</code>	Typecast (always performed)
<code>static_cast <type>(Value)</code>	Typecast (always performed)
<code>reinterpret_cast <type>(Value)</code>	Typecast (always performed)
<code>const_cast <type>(Value)</code>	Typecast (always performed)

Value operations

Operator	Meaning
<code>(type) Value</code>	Typecast (always performed)
<code>sizeof value</code>	Size of expression
<code>sizeof(type)</code>	Size of data type
<code>++ LValue</code>	Increment (before evaluation)
<code>-- LValue</code>	Decrement (before evaluation)
<code>~ Value</code>	Bit complement
<code>! Value</code>	Not (Boolean)

Operator	Meaning
<i>Value</i>	Unary minus
<i>+ Value</i>	Unary plus
<i>& LValue</i>	Address of data type
<i>Value</i>	Dereference
<i>Structure . Pointer</i>	Pointer to member of structure
<i>Pointer -> * Pointer</i>	Pointer to member of referenced structure

Arithmetic

Operator	Meaning
<i>Value Value</i>	Multiplication
<i>Value / Value</i>	Division
<i>Value % Value</i>	Modulus
<i>Value + Value</i>	Addition
<i>Value - Value</i>	Subtraction
<i>Value << Value</i>	Bitwise shift left
<i>Value >> Value</i>	Bitwise shift right
<i>Value < Value</i>	Less than (comparison)
<i>Value <= Value</i>	Less than or equal (comparison)
<i>Value > Value</i>	Greater than (comparison)
<i>Value >= Value</i>	Greater than or equal (comparison)
<i>Value == Value</i>	Equal (comparison)
<i>Value != Value</i>	Not equal (comparison)
<i>Value & Value</i>	Bitwise AND
<i>Value ^ Value</i>	Bitwise XOR (exclusive OR)
<i>Value Value</i>	Bitwise OR
<i>Value && Value</i>	Logical AND

Operator	Meaning
<code>Value Value</code>	Logical OR

The following examples assume that the pseudo registers are set as shown.

dbgcmd

```
0:000> r $t0 = 0
0:000> r $t1 = 1
0:000> r $t2 = 2
```

dbgcmd

```
0:000> ?? @$t1 + @$t2
unsigned int64 3
0:000> ?? @$t2/@$t1
unsigned int64 2
0:000> ?? @$t2|@$t1
unsigned int64 3
```

Assignment

Operator	Meaning
<code>LValue = Value</code>	Assign
<code>LValue *= Value</code>	Multiply and assign
<code>LValue /= Value</code>	Divide and assign
<code>LValue %= Value</code>	Modulo and assign
<code>LValue += Value</code>	Add and assign
<code>LValue -= Value</code>	Subtract and assign
<code>LValue <<= Value</code>	Shift left and assign
<code>LValue >>= Value</code>	Shift right and assign
<code>LValue &= Value</code>	AND and assign
<code>LValue = Value</code>	OR and assign
<code>LValue ^= Value</code>	XOR and assign

Evaluation

Operator	Meaning
<i>Value</i> ? <i>Value</i> : <i>Value</i>	Conditional evaluation
<i>Value</i> , <i>Value</i>	Evaluate all values, and then discard all except the rightmost value

Macros in C++ expressions

You can use macros within C++ expressions. You must add a number sign (#) before the macros.

You can use the following macros. These macros have the same definitions as the Microsoft Windows macros with the same name. The Windows macros are defined in `Winnt.h`.

Macro	Return value
<code>#CONTAINING_RECORD(Address, Type, Field)</code>	Returns the base address of an instance of a structure, given the type of the structure and the address of a field within the structure.
<code>#FIELD_OFFSET(Type, Field)</code>	Returns the byte offset of a named field in a known structure type.
<code>#RTL_CONTAINS_FIELD(Struct, Size, Field)</code>	Indicates whether the given byte size includes the desired field.
<code>#RTL_FIELD_SIZE(Type, Field)</code>	Returns the size of a field in a structure of known type, without requiring the type of the field.
<code>#RTL_NUMBER_OF(Array)</code>	Returns the number of elements in a statically sized array.
<code>#RTL_SIZEOF_THROUGH_FIELD(Type, Field)</code>	Returns the size of a structure of known type, up through and including a specified field.

This example shows the use of the `#FIELD_OFFSET` macro to calculate the byte offset to a field in a structure.

```
dbgcmd  
0:000> ?? #FIELD_OFFSET(_PEB, BeingDebugged)  
long 0n2
```

See also

[MASM expressions vs. C++ expressions](#)

[?? evaluate C++ expression](#)

[? evaluate expression](#)

[.expr choose expression evaluator](#)

[Sign extension](#)

[Mixed expression examples](#)

MASM Expressions vs. C++ Expressions

Article • 05/23/2024

The most significant differences between MASM expression evaluation and C++ expression evaluation are as follows:

- In a MASM expression, the numeric value of any symbol is its memory address. In a C++ expression, the numeric value of a variable is its actual value, not its address. Data structures do not have numeric values. Instead, they are treated as actual structures and you must use them accordingly. The value of a function name or any other entry point is the memory address and is treated as a function pointer. If you use a symbol that does not correspond to a C++ data type (such as an unmodified module name), a syntax error occurs.
- The MASM expression evaluator treats all numbers as ULONG64 values. The C++ expression evaluator casts numbers to ULONG64 and preserves type information of all data types.
- The MASM expression evaluator lets you to use any operator together with any number. The C++ expression evaluator generates an error if you use an operator together with an incorrect data type.
- In the MASM expression evaluator, all arithmetic is performed literally. In the C++ expression evaluator, pointer arithmetic is scaled properly and is not permitted when inappropriate.
- An MASM expression can use two underscores (__) or two colons (::) to indicate members of a class. The C++ expression evaluator uses only the two-colon syntax. Debugger *output* always uses two colons.
- In a MASM expression, you should add an at sign (@) before all except the most common registers. If you omit this at sign, the register name might be interpreted as a hexadecimal number or as a symbol. In a C++ expression, this prefix is required for all registers.
- MASM expressions might contain references to source lines. These references are indicated by grave accents (`). You cannot reference source line numbers in a C++ expression.

See also

[MASM Numbers and Operators](#)

[C++ Numbers and Operators](#)

[Mixed Expression Examples](#)

[Sign Extension](#)

Feedback

Was this page helpful?

 Yes

 No

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

Mixed Expression Examples

Article • 10/25/2023

This topics contains examples of MASM and C++ expressions that are used in various commands.

All other sections of this Help documentation use MASM expression syntax in the examples (unless otherwise noted). C++ expression syntax is very useful for manipulating structures and variables, but it does not parse the parameters of debugger commands very well.

If you are using debugger commands for general purposes or using debugger extensions, you should set MASM expression syntax as the default syntax, for example by using [.expr \(Choose Expression Evaluator\)](#). If you must have a specific parameter to use C++ expression syntax, use the @@() syntax.

If **myInt** is a ULONG32 value and if you are using the MASM expression evaluator, the following two examples show the value of **MyInt**.

```
dbgcmd  
0:000> ?? myInt  
0:000> dd myInt L1
```

However, the following example shows the *address* of **myInt**.

```
dbgcmd  
0:000> ? myInt
```

Conditional Breakpoints

You can use comparison operators to create [conditional breakpoints](#). The following code example uses MASM expression syntax. Because the current default radix is 16, the example uses the **On** prefix so that the number 20 is understood as a decimal number.

```
dbgcmd  
0:000> bp MyFunction+0x43 "j ( poi(MyVar)>0n20 ) ''; 'gc' "
```

In the previous example, **MyVar** is an integer in the C source. Because the MASM parser treats all symbols as addresses, the example must have the **poi** operator to dereference

MyVar.

Conditional Expressions

The following example prints the value of **ecx** if **eax** is greater than **ebx**, 7 if **eax** is less than **ebx**, and 3 if **eax** equals **ebx**. This example uses the MASM expression evaluator, so the equal sign (=) is a comparison operator, not an assignment operator.

```
dbgcmd
```

```
0:000> ? ecx*(eax>ebx) + 7*(eax<ebx) + 3*(eax==ebx)
```

In C++ syntax, the @ sign indicates a register, a double equal sign (==) is the comparison operator, and code must explicitly cast from **BOOL** to **int**. Therefore, in C++ syntax, the previous command becomes the following.

```
dbgcmd
```

```
0:000> ?? @ecx*(int)(@eax>@ebx) + 7*(int)(@eax<@ebx) + 3*(int)(@eax==@ebx)
```

MASM and C++ Mixed Expression Examples

You cannot use source-line expressions in a C++ expression. The following example uses the @@() alternative evaluator syntax to nest an MASM expression within a C++ expression. This example sets **MyPtr** equal to the address of line 43 of the **Myfile.c** file.

```
dbgcmd
```

```
0:000> ?? MyPtr = @@(`myfile.c:43` )
```

The following examples set the default expression evaluator to MASM and then evaluate *Expression2* as a C++ expression, and evaluate *Expression1* and *Expression3* as MASM expressions.

```
dbgcmd
```

```
0:000> .expr /s masm
0:000> bp Expression1 + @@( Expression2 ) + Expression3
```

If **myInt** is a **ULONG64** value and if you know that this value is followed in memory by another **ULONG64**, you can set an access breakpoint at that location by using one of the following examples. (Note the use of pointer arithmetic.)

```
dbgcmd
```

```
0:000> ba r8 @@( &myInt + 1 )
0:000> ba r8 myInt + 8
```

See also

[MASM Numbers and Operators](#)

[C++ Numbers and Operators](#)

[MASM Expressions vs. C++ Expressions](#)

[Sign Extension](#)

Sign Extension

Article • 10/25/2023

When a 32-bit signed integer is negative, its highest bit is equal to one. When this 32-bit signed integer is cast to a 64-bit number, the high bits can be set to zero (preserving the unsigned integer and hexadecimal value of the number) or the high bits can be set to one (preserving the signed value of the number). The latter situation is called *sign extension*.

The debugger follows different rules for sign extension in MASM expressions, in C++ expressions, and when displaying numbers.

Sign Extension in MASM Expressions

Under certain conditions, numbers are automatically *sign extended* by the MASM expression evaluator. Sign extension can affect only numbers from 0x80000000 through 0xFFFFFFFF. That is, sign extension affects only numbers that can be written in 32 bits with the high bit equal to 1.

The number 0x12345678 always remains 0x00000000`12345678 when the debugger treats it as a 64-bit number. On the other hand, when 0x890ABCDE is treated as a 64-bit value, it might remain 0x00000000`890ABCDE or the MASM expression evaluator might sign extend it to 0xFFFFFFFF`890ABCDE.

A number from 0x80000000 through 0xFFFFFFFF is sign extended based on the following criteria:

- Numeric constants are never sign extended in user mode. In kernel mode, a numeric constant is sign extended unless it contains a grave accent (`) before the low bytes. For example, in kernel mode, the hexadecimal numbers **EEAA1122** and **00000000EEAA1122** are sign extended, but **00000000`EEAA1122** and **0`EEAA1122** are not.
- A 32-bit register is sign extended in both modes.
- Pseudo-registers are always stored as 64-bit values. They are not sign extended when they are evaluated. When a pseudo-register is *assigned* a value, the expression that is used is evaluated according to the standard C++ criteria.
- Individual numbers and registers in an expression can be sign extended, but no other calculations during expression evaluation are sign extended. As a result, you can mask the high bits of a number or register by using the following syntax.

Console

```
( 0x0`FFFFFF & expression )
```

Sign Extension in C++ Expressions

When the debugger evaluates a C++ expression, the following rules apply:

- Registers and pseudo-registers are never sign extended.
- All other values are treated exactly like C++ would treat values of their type.

Displaying Sign-Extended and 64-Bit Numbers

Other than 32-bit and 16-bit registers, all numbers are stored internally within the debugger as 64-bit values. However, when a number satisfies certain criteria, the debugger displays it as a 32-bit number in command output.

The debugger uses the following criteria to determine how to display numbers:

- If the high 32 bits of a number are all zeros (that is, if the number is from 0x00000000`00000000 through 0x00000000`FFFFFF), the debugger displays the number as a 32-bit number.
- If the high 32 bits of a number are all ones and if the highest bit of the low 32 bits is also a one (that is, if the number is from 0xFFFFFFF`80000000 through 0xFFFFFFF`FFFFFFF), the debugger assumes the number is a sign-extended 32-bit number and displays it as a 32-bit number.
- If the previous two condition do not apply (that is, if the number is from 0x00000001`00000000 through 0xFFFFFFF`7FFFFFF) the debugger displays the number as a 64-bit number.

Because of these display rules, when a number is displayed as a 32-bit number from 0x80000000 through 0xFFFFFFF, you cannot confirm whether the high 32 bits are all ones or all zeros. To distinguish between these two cases, you must perform an additional computation on the number (such as masking one or more of the high bits and displaying the result).

See also

[MASM Numbers and Operators](#)

C++ Numbers and Operators

MASM Expressions vs. C++ Expressions

Mixed Expression Examples

Sign Extension

String Wildcard Syntax

Article • 10/25/2023

Some debugger commands have string parameters that accept a variety of wildcard characters. These parameters are noted on their respective reference pages.

These kinds of parameters support the following syntax features:

- An asterisk (*) represents zero or more characters.
- A question mark (?) represents any single character.
- Brackets ([]) that contain a list of characters represent any single character in the list. Exactly one character in the list is matched. Within these brackets, you can use a hyphen (-) to specify a range. For example, **Prog[er-t7]am** matches "Progeam", "Program", "Progsam", "Progtam", and "Prog7am".
- A number sign (#) represents zero or more of the preceding characters. For example, **Lo#p** matches "Lp", "Lop", "Loop", "Looop", and so on. You can also combine a number sign with brackets, so **m[ia]#n** matches "mn", "min", "man", "maan", "main", "mian", "miin", "miain", and so on.
- A plus sign (+) represents one or more of the preceding characters. For example, **Lo+p** is the same as **Lo#p**, except that **Lo+p** does not match "Lp". Similarly, **m[ia]+n** is the same as **m[ia]#n**, except that **m[ia]+n** does not match "mn". **a?+b** is also the same as **a*b**, except that **a?+b** does not match "ab".
- If you have to specify a literal number sign (#), question mark (?), opening bracket ((), closing bracket ()), asterisk (*), or plus sign (+) character, you must add a backslash (\) in front of the character. Hyphens are always literal when you do not enclose them in brackets. But you cannot specify a literal hyphen within a bracketed list.

Parameters that specify symbols also support some additional features. In addition to the standard string wildcard characters, you can use an underscore (_) before a text expression that you use to specify a symbol. When matching this expression to a symbol, the debugger treats the underscore as any quantity of underscores, even zero. This feature applies only when you are matching symbols. It does not apply to string wildcard expressions in general. For more information about symbol syntax, see [Symbol Syntax and Symbol Matching](#).

Register Syntax

Article • 10/25/2023

The debugger can control registers and floating-point registers.

When you use a register in an expression, you should add an at sign (@) before the register. This at sign tells the debugger that the following text is the name of a register.

If you are using MASM expression syntax, you can omit the at sign for certain very common registers. On x86-based systems, you can omit the at sign for the **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**, **eip**, and **efl** registers. However, if you specify a less common register without an at sign, the debugger first tries to interpret the text as a hexadecimal number. If the text contains non-hexadecimal characters, the debugger next interprets the text as a symbol. Finally, if the debugger does not find a symbol match, the debugger interprets the text as a register.

If you are using C++ expression syntax, the at sign is always required.

The **r (Registers)** command is an exception to this rule. The debugger always interprets its first argument as a register. (An at sign is not required or permitted.) If there is a second argument for the **r** command, it is interpreted according to the default expression syntax. If the default expression syntax is C++, you must use the following command to copy the **ebx** register to the **eax** register.

```
dbgcmd  
0:000> r eax = @ebx
```

For more information about the registers and instructions that are specific to each processor, see [Processor Architecture](#).

Flags on an x86-based Processor

x86-based processors also use several 1-bit registers known as *flags*. For more information about these flags and the syntax that you can use to view or change them, see [x86 Flags](#).

Registers and Threads

Each thread has its own register values. These values are stored in the CPU registers when the thread is executing and in memory when another thread is executing.

In user mode, any reference to a register is interpreted as the register that is associated with the current thread. For more information about the current thread, see [Controlling Processes and Threads](#).

In kernel mode, any reference to a register is interpreted as the register that is associated with the current register context. You can set the register context to match a specific thread, context record, or trap frame. You can display only the most important registers for the specified register context, and you cannot change their values.

Pseudo-Register Syntax

Article • 10/25/2023

The debugger supports several pseudo-registers that hold certain values.

The debugger sets *automatic pseudo-registers* to certain useful values. *User-defined pseudo-registers* are integer variables that you can write to or read.

All pseudo-registers begin with a dollar sign (\$). If you are using MASM syntax, you can add an at sign (@) before the dollar sign. This at sign tells the debugger that the following token is a register or pseudo-register, not a symbol. If you omit the at sign, the debugger responds more slowly, because it has to search the whole symbol table.

For example, the following two commands produce the same output, but the second command is faster.

```
dbgcmd  
  
0:000> ? $exp  
Evaluate expression: 143 = 0000008f  
0:000> ? @$exp  
Evaluate expression: 143 = 0000008f
```

If a symbol exists with the same name as the pseudo-register, you must add the at sign.

If you are using C++ expression syntax, the at sign (@) is always required.

The [r \(Registers\)](#) command is an exception to this rule. The debugger always interprets its first argument as a register or pseudo-register. (An at sign is not required or permitted.) If there is a second argument for the r command, it is interpreted according to the default expression syntax. If the default expression syntax is C++, you must use the following command to copy the \$t2 pseudo-register to the \$t1 pseudo-register.

```
dbgcmd  
  
0:000> r $t1 = @$t2
```

Automatic Pseudo-Registers

The debugger automatically sets the following pseudo-registers.

Pseudo-register	Description
\$ea	The effective address of the last instruction that was executed. If this instruction does not have an effective address, the debugger displays "Bad register error". If this instruction has two effective addresses, the debugger displays the first address.
\$ea2	The second effective address of the last instruction that was executed. If this instruction does not have two effective addresses, the debugger displays "Bad register error".
\$exp	The last expression that was evaluated.
\$ra	The return address that is currently on the stack. This address is especially useful in execution commands. For example, <code>g @\$ra</code> continues until the return address is found (although gu (Go Up) is a more precise effective way of "stepping out" of the current function).
\$ip	The instruction pointer register. x86-based processors: The same as <code>eip</code> . Itanium-based processors: Related to <code>iip</code> . (For more information, see the note following this table.) x64-based processors: The same as <code>rip</code> .
\$eventip	The instruction pointer at the time of the current event. This pointer typically matches <code>\$ip</code> , unless you switched threads or manually changed the value of the instruction pointer.
\$previp	The instruction pointer at the time of the previous event. (Breaking into the debugger counts as an event.)
\$relip	An instruction pointer that is related to the current event. When you are branch tracing, this pointer is the pointer to the branch source.
\$scopeip	The instruction pointer for the current local context (also known as the <code>scope</code>).
\$exentry	The address of the entry point of the first executable of the current process.
\$retreg	The primary return value register.

Pseudo-register	Description x86-based processors: The same as eax . Itanium-based processors: The same as ret0 . x64-based processors: The same as rax .
\$retreg64	The primary return value register, in 64-bit format. x86 processor: The same as the edx:eax pair.
\$csp	The current call stack pointer. This pointer is the register that is most representative of call stack depth. x86-based processors: The same as esp . Itanium-based processors: The same as bsp . x64-based processors: The same as rsp .
\$p	The value that the last d* (Display Memory) command printed.
\$proc	The address of the current process (that is, the address of the EPROCESS block).
\$thread	The address of the current thread. In kernel-mode debugging, this address is the address of the ETHREAD block. In user-mode debugging, this address is the address of the thread environment block (TEB).
\$peb	The address of the process environment block (PEB) of the current process.
\$teb	The address of the thread environment block (TEB) of the current thread.
\$tpid	The process ID (PID) for the process that owns the current thread.
\$tid	The thread ID for the current thread.
\$dtid	
\$dpid	
\$dsid	
\$bp <i>Number</i>	The address of the corresponding breakpoint. For example, \$bp3 (or \$bp03) refers to the breakpoint whose breakpoint ID is 3. <i>Number</i> is always a decimal number. If no breakpoint has an ID of <i>Number</i> , \$bp <i>Number</i> evaluates to

Pseudo-register	Description
	zero. For more information about breakpoints, see Using Breakpoints .
\$frame	The current frame index. This index is the same frame number that the .frame (Set Local Context) command uses.
\$dbgtime	The current time, according to the computer that the debugger is running on.
\$callret	The return value of the last function that .call (Call Function) called or that is used in an .fnret /s command. The data type of \$callret is the data type of this return value.
\$extret	
\$extin	
\$clrex	
\$lastclrex	Managed debugging only: The address of the last-encountered common language runtime (CLR) exception object.
\$ptrsize	The size of a pointer. In kernel mode, this size is the pointer size on the target computer.
\$pagesize	The number of bytes in one page of memory. In kernel mode, this size is the page size on the target computer.
\$pcr	
\$pcrb	
\$argreg	
\$exr_chance	The chance of the current exception record.
\$exr_code	The exception code for the current exception record.
\$exr_numparams	The number of parameters in the current exception record.
\$exr_param0	The value of Parameter 0 in the current exception record.
\$exr_param1	The value of Parameter 1 in the current exception record.

Pseudo-register	Description
	exception record.
\$exr_param2	The value of Parameter 2 in the current exception record.
\$exr_param3	The value of Parameter 3 in the current exception record.
\$exr_param4	The value of Parameter 4 in the current exception record.
\$exr_param5	The value of Parameter 5 in the current exception record.
\$exr_param6	The value of Parameter 6 in the current exception record.
\$exr_param7	The value of Parameter 7 in the current exception record.
\$exr_param8	The value of Parameter 8 in the current exception record.
\$exr_param9	The value of Parameter 9 in the current exception record.
\$exr_param10	The value of Parameter 10 in the current exception record.
\$exr_param11	The value of Parameter 11 in the current exception record.
\$exr_param12	The value of Parameter 12 in the current exception record.
\$exr_param13	The value of Parameter 13 in the current exception record.
\$exr_param14	The value of Parameter 14 in the current exception record.
\$bug_code	If a bug check has occurred, this is the bug code. Applies to live kernel-mode debugging and kernel crash dumps.
\$bug_param1	If a bug check has occurred, this is the value of Parameter 1. Applies to live kernel-mode debugging and kernel crash dumps.
\$bug_param2	If a bug check has occurred, this is the value of Parameter 2. Applies to live kernel-mode

Pseudo-register	Description
	debugging and kernel crash dumps.
\$bug_param3	If a bug check has occurred, this is the value of Parameter 3. Applies to live kernel-mode debugging and kernel crash dumps.
\$bug_param4	If a bug check has occurred, this is the value of Parameter 4. Applies to live kernel-mode debugging and kernel crash dumps.

Some of these pseudo-registers might not be available in certain debugging scenarios. For example, you cannot use **\$peb**, **\$tid**, and **\$tpid** when you are debugging a user-mode minidump or certain kernel-mode dump files. There will be situations where you can learn thread information from [~ \(Thread Status\)](#) but not from **\$tid**. You cannot use the **\$previp** pseudo-register on the first debugger event. You cannot use the **\$relip** pseudo-register unless you are branch tracing. If you use an unavailable pseudo-register, a syntax error occurs.

A pseudo-register that holds the address of a structure -- such as **\$thread**, **\$proc**, **\$teb**, **\$peb**, and **\$lastClrex** -- will be evaluated according to the proper data type in the C++ expression evaluator, but not in the MASM expression evaluator. For example, the command **? \$teb** displays the address of the TEB, while the command **?? @\$teb** displays the entire TEB structure. For more information, see [Evaluating Expressions](#).

On an Itanium-based processor, the **iip** register is *bundle-aligned*, which means that it points to slot 0 in the bundle containing the current instruction, even if a different slot is being executed. So **iip** is not the full instruction pointer. The **\$ip** pseudo-register is the actual instruction pointer, including the bundle and the slot. The other pseudo-registers that hold address pointers (**\$ra**, **\$retreg**, **\$eventip**, **\$previp**, **\$relip**, and **\$exentry**) have the same structure as **\$ip** on all processors.

You can use the **r** command to change the value of **\$ip**. This change also automatically changes the corresponding register. When execution resumes, it resumes at the new instruction pointer address. This register is the only automatic pseudo-register that you can change manually.

Note In MASM syntax, you can indicate the **\$ip** pseudo-register with a period (.). You do not add an at sign (@) before this period, and do not use the period as the first parameter of the **r** command. This syntax is not permitted within a C++ expression.

Automatic pseudo-registers are similar to [automatic aliases](#). But you can use automatic aliases together with alias-related tokens (such as **{ }**), and you cannot use pseudo-registers with such tokens.

User-Defined Pseudo-Registers

There are 20 user-defined pseudo-registers (`$t0`, `$t1`, ..., `$t19`). These pseudo-register are variables that you can read and write through the debugger. You can store any integer value in these pseudo-registers. They can be especially useful as loop variables.

To write to one of these pseudo-registers, use the [r \(Registers\)](#) command, as the following example shows.

```
dbgcmd  
0:000> r $t0 = 7  
0:000> r $t1 = 128*poi(MyVar)
```

Like all pseudo-registers, you can use the user-defined pseudo-register in any expression, as the following example shows.

```
dbgcmd  
0:000> bp $t3  
0:000> bp @$t4  
0:000> ?? @$t1 + 4*@$t2
```

A pseudo-register is always typed as an integer, unless you use the `? switch` together with the `r` command. If you use this switch, the pseudo-register acquires the type of whatever is assigned to it. For example, the following command assigns the `UNICODE_STRING**` type and the `0x0012FFBC` value to `$t15`.

```
dbgcmd  
0:000> r? $t15 = * (UNICODE_STRING*) 0x12ffbc
```

User-defined pseudo-registers use zero as the default value when the debugger is started.

Note The aliases `$u0`, `$u1`, ..., `$u9` are not pseudo-registers, despite their similar appearance. For more information about these aliases, see [Using Aliases](#).

Example

The following example sets a breakpoint that is hit every time that the current thread calls `NtOpenFile`. But this breakpoint is not hit when other threads call `NtOpenFile`.

```
dbgcmd
```

```
kd> bp /t @$thread nt!ntopenfile
```

Example

The following example executes a command until the register holds a specified value. First, put the following code for conditional stepping in a script file named "eaxstep".

```
dbgcmd
```

```
.if (@eax == 1234) { .echo 1234 } .else { t "$<eaxstep" }
```

Next, issue the following command.

```
dbgcmd
```

```
t "$<eaxstep"
```

The debugger performs a step and then runs your command. In this case, the debugger runs the script, which either displays **1234** or repeats the process.

Source Line Syntax

Article • 05/23/2024

You can specify source file line numbers as all or part of a MASM expression. These numbers evaluate to the offset of the executable code that corresponds to this source line.

Note You cannot use source line numbers as part of a C++ expression. For more information about when MASM and C++ expression syntax is used, see [Evaluating Expressions](#).

You must enclose source file and line number expressions by grave accents (`). The following example shows the full format for source file line numbers.

```
text
`[[Module!]Filename][:LineNumber]`
```

If you have multiple files that have identical file names, *Filename* should include the whole directory path and file name. This directory path should be the one that is used at compilation time. If you supply only the file name or only part of the path and if there are multiple matches, the debugger uses the first match that it finds.

If you omit *Filename*, the debugger uses the source file that corresponds to the current program counter.

LineNumber is read as a decimal number unless you precede it with **0x**, regardless of the current default radix. If you omit *LineNumber*, the expression evaluates to the initial address of the executable that corresponds to the source file.

Source line expressions are not evaluated in CDB unless you issue a [.lines \(Toggle Source Line Support\)](#) command or you include the [-lines command-line option](#) when you start WinDbg..

For more information about source debugging, see [Debugging in Source Mode](#).

Feedback

Was this page helpful?



Yes



No

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

Address and Address Range Syntax

Article • 10/25/2023

There are several ways to specify addresses in the debugger.

Addresses are normally *virtual addresses*, except when the documentation specifically indicates another kind of address. In user mode, the debugger interprets virtual addresses according to the page directory of the [current process](#). In kernel mode, the debugger interprets virtual addresses according to the page directory of the process that the [process context](#) specifies. You can also directly set the *user-mode address context*. For more information about the user-mode address context, see [.context \(Set User-Mode Address Context\)](#).

In MASM expressions, you can use the **poi** operator to dereference any pointer. For example, if the pointer at address 0x0000008e`ed57b108 points to address location 0x805287637256, the following two commands are equivalent.

```
dbgcmd  
0:000> dd 805287637256  
0:000> dd poi(000000bb`7ee23108)
```

Display memory address example

To see an example of using **poi**, determine the offset for the *CurrentLocale* of the thread environment block (TEB). Use the **dx** command to display **@\$teb**, which is an example of a [pseudo-registers](#), that hold common addresses, such as the current program counter location.

```
dbgcmd  
0:000> dx @$teb  
@$teb : 0x1483181000 [Type: _TEB *]  
...  
[+0x108] CurrentLocale : 0x409 [Type: unsigned long]
```

CurrentLocale is +0x108 from the start of the TEB. Next determine the memory address of that location.

```
dbgcmd
```

```
0:000> ? @$teb + 0x108  
Evaluate expression: 613867303176 = 0000008e`ed57b108
```

Use `poi` to dereference that address to see that it contains the `CurrentLocale` value of `0x409`.

```
dbgcmd
```

```
0:000> ? poi(0000008e`ed57b108)  
Evaluate expression: 1033 = 00000000`00000409
```

In C++ debugger expressions, pointers behave like pointers in C++. However, numbers are interpreted as integers. If you have to deference an actual number, you may need to cast it first, as the following example shows.

To try this, use `.expr` to set the expression evaluator to C++.

```
dbgcmd
```

```
0:000> .expr /s C++  
Current expression evaluator: C++ - C++ source expressions
```

With the expression evaluator set to C++, we can cast using `long`.

```
dbgcmd
```

```
0:000> d *((long*)0x00000014`83181108 )  
00000000`00000409 ??????? ??????? ??????? ???????
```

For more information about casting numeric values, see [C++ Numbers and Operators](#).

If the expression evaluator is set to `c++`, we can wrap the `poi` pointer with `@@masm()`, to have just that part of the expression evaluated by the MASM expression evaluator.

```
dbgcmd
```

```
0:000> .expr /s c++  
Current expression evaluator: C++ - C++ source expressions  
  
0:000> ? @@masm(poi(00000078`267d7108))  
Evaluate expression: 1033 = 00000000`00000409
```

For more information about the two expression evaluators, see [Evaluating Expressions](#).

You can also indicate an address in an application by specifying the original source file name and line number. For more information about how to specify this information, see [Source Line Syntax](#).

Address Ranges

You can specify an address range by a pair of addresses or by an address and object count.

To specify a range by a pair of addresses, specify the starting address and the ending address. For example, the following example is a range of 8 bytes, beginning at the address 0x00001000.

```
dbgcmd  
0x00001000 0x00001007
```

To specify an address range by an address and object count, specify an address argument, the letter L (uppercase or lowercase), and a value argument. The address specifies the starting address. The value specifies the number of objects to be examined or displayed. The size of the object depends on the command. For example, if the object size is 1 byte, the following example is a range of 8 bytes, beginning at the address 0x00001000.

```
dbgcmd  
0x00001000 L8
```

However, if the object size is a double word (32 bits or 4 bytes), the following two ranges each give an 8-byte range.

```
dbgcmd  
0x00001000 0x00001007  
0x00001000 L2
```

L Size range specifier

There are two other ways to specify the value (the **LSize** range specifier):

- **L? Size** (with a question mark) means the same as **LSize**, except that **L? Size** removes the debugger's automatic range limit. Typically, there is a range limit of

256 MB, because larger ranges are typographic errors. If you want to specify a range that is larger than 256 MB, you must use the **L? Size** syntax.

- **L- Size** (with a hyphen) specifies a range of length *Size* that ends at the given address. For example, **80000000 L20** specifies the range from 0x80000000 through 0x8000001F, and **80000000 L-20** specifies the range from 0x7FFFFFFE0 through 0x7FFFFFFF.

Some commands that ask for address ranges accept a single address as the argument. In this situation, the command uses some default object count to compute the size of the range. Typically, commands for which the address range is the final parameter permit this syntax. For the exact syntax and the default range size for each command, see the reference topics for each command.

Search memory range example

First we will determine the address of the rip instruction pointer register using the MASM expression evaluator.

```
dbgcmd

0:000> ? @rip
Evaluate expression: 140720561719153 = 00007ffc`0f180771
```

Then we will search starting at 00007ffc`0f180771, for 100000 using the [s \(Search Memory\)](#) command. We specify the range to search using L100000.

```
dbgcmd

0:000> s -a 00007ffc`0f180771 L100000 "ntdll"
00007ffc`0f1d48fa 6e 74 64 6c 6c 5c 6c 64-72 69 6e 69 74 2e 63 00
ntdll\ldrinit.c.
00007ffc`0f1d49c2 6e 74 64 6c 6c 5c 6c 64-72 6d 61 70 2e 63 00 00
ntdll\ldrmap.c..
00007ffc`0f1d4ab2 6e 74 64 6c 6c 5c 6c 64-72 72 65 64 69 72 65 63
ntdll\ldrredirec
00007ffc`0f1d4ad2 6e 74 64 6c 6c 5c 6c 64-72 73 6e 61 70 2e 63 00
ntdll\ldrsnap.c.
...
```

We can also specify the same range like this using two memory addresses.

```
dbgcmd
```

```
0:000> s -a 0x00007ffc`0f180771 0x00007ffc`0f280771 "ntdll"
00007ffc`0f1d48fa 6e 74 64 6c 6c 5c 6c 64-72 69 6e 69 74 2e 63 00
ntdll\ldrinit.c.
00007ffc`0f1d49c2 6e 74 64 6c 6c 5c 6c 64-72 6d 61 70 2e 63 00 00
ntdll\ldrmap.c..
00007ffc`0f1d4ab2 6e 74 64 6c 6c 5c 6c 64-72 72 65 64 69 72 65 63
ntdll\ldrredirec
00007ffc`0f1d4ad2 6e 74 64 6c 6c 5c 6c 64-72 73 6e 61 70 2e 63 00
ntdll\ldrsnap.c.
...
...
```

Lastly we can search backwards in the memory range by using the L- length parameter.

```
dbgcmd

0:000> s -a 00007ffc`0f1d4ad2 L-100000 "ntdll"
00007ffc`0f1d48fa 6e 74 64 6c 6c 5c 6c 64-72 69 6e 69 74 2e 63 00
ntdll\ldrinit.c.
00007ffc`0f1d49c2 6e 74 64 6c 6c 5c 6c 64-72 6d 61 70 2e 63 00 00
ntdll\ldrmap.c..
00007ffc`0f1d4ab2 6e 74 64 6c 6c 5c 6c 64-72 72 65 64 69 72 65 63
ntdll\ldrredirec
```

Unassemble memory example

This example, uses the [u \(unassemble\)](#) command and the L parameter to unassemble three bytes of code.

```
dbgcmd

0:000> u 00007ffc`0f1d48fa L3
ntdll!`string'+0xa:
00007ffc`0f1d48fa 6e          outs    dx,byte ptr [rsi]
00007ffc`0f1d48fb 7464        je      ntdll!`string'+0x21
(00007ffc`0f1d4961)
00007ffc`0f1d48fd 6c          ins     byte ptr [rdi],dx
```

Or specify a three byte range of memory to unassemble like this.

```
dbgcmd

0:000> u 00007ffc`0f1d48fa 00007ffc`0f1d48fd
ntdll!`string'+0xa:
00007ffc`0f1d48fa 6e          outs    dx,byte ptr [rsi]
00007ffc`0f1d48fb 7464        je      ntdll!`string'+0x21
```

```
(00007ffc`0f1d4961)
00007ffc`0f1d48fd 6c
```

```
ins     byte ptr [rdi],dx
```

Address Modes and Segment Support

On x86-based platforms, CDB and KD support the following addressing modes. These modes are distinguished by their prefixes.

Prefix	Name	Address types
%	flat	32-bit addresses (also 16-bit selectors that point to 32-bit segments) and 64-bit addresses on 64-bit systems.
&	virtual 86	Real-mode addresses. x86-based only.
#	plain	Real-mode addresses. x86-based only.

The difference between the plain and virtual 86 modes is that a plain 16-bit address uses the segment value as a selector and looks up the segment descriptor. But a virtual 86 address does not use selectors and instead maps directly into the lower 1 MB.

If you access memory through an addressing mode that is not the current default mode, you can use the address mode prefixes to override the current address mode.

Address Arguments

Address arguments specify the location of variables and functions. The following table explains the syntax and meaning of the various addresses that you can use in CDB and KD.

Syntax	Meaning
offset	The absolute address in virtual memory space, with a type that corresponds to the current execution mode. For example, if the current execution mode is 16 bit, the offset is 16 bit. If the execution mode is 32-bit segmented, the offset is 32-bit segmented.
&[[segment:]] offset	The real address. x86-based and x64-based.
%segment:[offset]	A segmented 32-bit or 64-bit address. x86-based and x64-based.

Syntax	Meaning
<code>%[[offset]]</code>	An absolute address (32-bit or 64-bit) in virtual memory space. x86-based and x64-based.
<code>name[[+ −]] offset</code>	A flat 32-bit or 64-bit address. <i>name</i> can be any symbol. <i>offset</i> specifies the offset. This offset can be whatever address mode its prefix indicates. No prefix specifies a default mode address. You can specify the offset as a positive (+) or negative (−) value.

Use the [dg \(Display Selector\)](#) command to view segment descriptor information.

See Also

To display information about memory, use the [!address](#) command.

To search memory, use the [s \(Search Memory\)](#) command.

To display the contents of memory use the [d, da, db, dc, dd, dD, df, dp, dq, du, dw \(Display Memory\)](#) command.

For information on how you can view and edit memory using a Memory window see [Using a Memory Window](#).

Thread Syntax

Article • 10/25/2023

Many debugger commands have thread identifiers as their parameters. A tilde (~) appears before the thread identifier.

The thread identifier can be one of the following values.

Thread identifier	Description
~.	The current thread.
~#	The thread that caused the current exception or debug event.
~*	All threads in the process.
~Number	The thread whose index is <i>Number</i> .
~~[TID]	The thread whose thread ID is <i>TID</i> . (The brackets are required And you cannot add a space between the second tilde and the opening bracket.)
~[Expression]	The thread whose thread ID is the integer to which the numerical <i>Expression</i> resolves.

Threads are assigned indexes as they are created. Note that this number differs from the thread ID that the Microsoft Windows operating system uses.

When debugging begins, the current thread is the one that caused the present exception or debug event (or the active thread when the debugger attached to the process). That thread remains the current thread until you specify a new one by using a [~s \(Set Current Thread\)](#) command or by using the [Processes and Threads window](#) in WinDbg.

Thread identifiers typically appear as command prefixes. Note that not all wildcard characters are available in all commands that use thread identifiers.

An example of the ~[Expression] syntax would be `~[@$t0]`. In this example, the thread changes depending on the value of a user-defined pseudo-register. This syntax allows debugger scripts to programmatically select a thread.

Controlling Threads in Kernel Mode

In kernel mode, you cannot control threads by using thread identifiers. For more information about how to access thread-specific information in kernel mode, see [Changing Contexts](#).

Note You can use the tilde character (~) to specify threads during user-mode debugging. In kernel-mode debugging, you can use the tilde to specify processors. For more information about how to specify processors, see [Multiprocessor Syntax](#).

Process Syntax

Article • 10/25/2023

Many debugger commands have process identifiers as their parameters. A vertical bar (|) appears before the process identifier.

The process identifier can be one of the following values.

Process identifier	Description
.	The current process.
#	The process that caused the current exception or debug event.
*	All processes.
Number	The process whose ordinal is <i>Number</i> .
~[PID]	The process whose process ID is <i>PID</i> . (The brackets are required and you cannot add a space between the tilde (~) and the opening bracket.)
[Expression]	The process whose process ID is the integer to which the numerical <i>Expression</i> resolves.

Processes are assigned ordinals as they are created. Note that this number differs from the process ID (PID) that the Microsoft Windows operating system uses.

The current process defines the memory space and the set of threads that are used. When debugging begins, the current process is the one that caused the present exception or debug event (or the process that the debugger attached to). That process remains the current process until you specify a new one by using a [|s \(Set Current Process\)](#) command or by using the [Processes and Threads window](#) in WinDbg.

Process identifiers are used as parameters in several commands, frequently as the command prefix. Note that WinDbg and CDB can debug child processes that the original process created. WinDbg and CDB can also attach to multiple unrelated processes.

An example of the |[Expression] syntax would be |[@\$t0]. In this example, the process changes depending on the value of a user-defined pseudo-register. This syntax allows debugger scripts to programmatically select a process.

Controlling Processes in Kernel Mode

In kernel mode, you cannot control processes by using process identifiers. For more information about how to access process-specific information in kernel mode, see [Changing Contexts](#).

System Syntax

Article • 10/25/2023

Many debugger commands have process identifiers as their parameters.

Two vertical bars (||) appear before the system identifier. The system identifier can be one of the following values.

System identifier	Description
.	The current system
#	The system that caused the current exception or debug event.
*	All systems.
<i>ddd</i>	The system whose ordinal is <i>ddd</i> .

Systems are assigned ordinals in the order that the debugger attaches to them.

When debugging begins, the current system is the one that caused the present exception or debug event (or the one that the debugger most recently attached to). That system remains the current system until you specify a new one by using a [||s \(Set Current System\)](#) command or by using the [Processes and Threads window](#) in WinDbg.

Example

This example shows three dump files are loaded. System 1 is active and system 2 caused the debug event.

```
dbgcmd

||1:1:017> ||
  0 User mini dump: c:\notepad.dmp
.  1 User mini dump: c:\paint.dmp
#  2 User mini dump: c:\calc.dmp
```

Remarks

To work with multiple systems, you can use the [.opendump](#) to debug multiple crash dumps at the same time. For more information about how to control a multiple-target

session, see [Debugging Multiple Targets](#).

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

Multiprocessor Syntax

Article • 10/25/2023

KD and kernel-mode WinDbg support multiple processor debugging. You can perform this kind of debugging on any multiprocessor platform.

Processors are numbered zero through n .

If the current processor is processor 0 (that is, if it is the processor that currently caused the debugger to be active), you can examine the other non-current processors (processors one through n). However, you cannot change anything in the non-current processors. You can only view their state.

Selecting a Processor

You can use the [.echocpnum \(Show CPU Number\)](#) command to display the processor numbers of the current processor. The output from this command enables you to immediately tell when you are working on a multiple processor system by the text in the kernel debugging prompt.

In the following example, 0: in front of the `kd>` prompt indicates that you are debugging the first processor in the computer.

```
dbgcmd
```

```
0: kd>
```

Use the [~s \(Change Current Processor\)](#) command to switch between processors, as the following example shows.

```
dbgcmd
```

```
0: kd> ~1s  
1: kd>
```

Now you are debugging the second processor in the computer.

You might have to change processors on a multiprocessor system if you encounter a break and you cannot understand the stack trace. The break might have occurred on a different processor.

Specifying Processors in Other Commands

You can add a processor number before several commands. This number is not preceded by a tilde (~), except in the ~S command.

Note In user-mode debugging, the tilde is used to specify threads. For more information about this syntax, see [Thread Syntax](#).

Processor IDs do not have to be referred to explicitly. Instead, you can use a numerical expression that resolves to an integer that corresponds to a processor ID. To indicate that the expression should be interpreted as a processor, use the following syntax.

```
dbgcmd  
| | [Expression]
```

In this syntax, the square brackets are required, and *Expression* stands for any numerical expression that resolves to an integer that corresponds to a processor ID.

In the following example, the processor changes depending on the value of a user-defined pseudo-register.

```
dbgcmd  
| | [@$t0]
```

Examples

The following example uses the [k \(Display Stack Backtrace\)](#) command to display a stack trace from processor two.

```
dbgcmd  
1: kd> 2k
```

The following example uses the [r \(Registers\)](#) command to display the eax register of processor three.

```
dbgcmd  
1: kd> 3r eax
```

However, the following command gives a syntax error, because you cannot change the state of a processor other than the current processor.

```
dbgcmd  
1: kd> 3r eax=808080
```

Breakpoints

During kernel debugging, the [bp](#), [bu](#), [bm](#) (**Set Breakpoint**) and [ba](#) (**Break on Access**) commands apply to all processors of a multiple processor computer.

For example, if the current processor is three, you can enter the following command to put a breakpoint at **SomeAddress**.

```
dbgcmd  
1: kd> bp SomeAddress
```

Then, any processor (not only processor one) that executes at that address causes a breakpoint trap.

Displaying Processor Information

You can use the [!running](#) extension to display the status of each processor on the target computer. For each processor, [!running](#) can also display the current and next thread fields from the process control block (PRCB), the state of the 16 built-in queued spinlocks, and a stack trace.

You can use the [!cpuinfo](#) and [!cpuid](#) extensions to display information about the processors themselves.

Command Tokens

Article • 01/23/2024

This section of the reference discusses the various tokens used within debugger commands and meta-commands.

These tokens include:

[; \(Command Separator\)](#)

[{ } \(Block Delimiter\)](#)

[\\${} \(Alias Interpreter\)](#)

[\\$\\$ \(Comment Specifier\)](#)

[* \(Comment Line Specifier\)](#)

[.block](#)

[.break ↗](#)

[.catch](#)

[.continue](#)

[.do](#)

[.else](#)

[.elsif](#)

[.for](#)

[.foreach](#)

[.if](#)

[.leave](#)

[.printf](#)

[.while](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

;(Command Separator)

Article • 10/25/2023

The semicolon (;) character is used to separate multiple commands on a single line.

```
dbgcmd
```

```
Command1 ; Command2 [; Command3 ...]
```

Parameters

Command1, Command2, ...

The commands to be executed.

Remarks

Commands are executed sequentially from left to right. All commands on a single line refer to the current thread, unless otherwise specified. If a command causes the thread to execute, the remaining commands on the line will be deferred until that thread stops on a debug event.

A small number of commands cannot be followed by a semicolon, because they automatically take the entire remainder of the line as their argument. These include [as \(Set Alias\)](#), [\\$<\(Run Script File\)](#), and any command beginning with the [* \(Comment Line Specifier\)](#) token.

Here is an example. This executes the current program to source line 123, prints the value of counter, then resumes execution:

```
dbgcmd
```

```
0:000> g `:123` ; ? poi(counter); g
```

{ } (Block Delimiter)

Article • 10/25/2023

A pair of braces ({ }) is used to surround a block of statements within a debugger command program.

```
dbgcmd
```

```
Statements { Statements } Statements
```

Additional Information

For information about debugger command programs and control flow tokens, see [Using Debugger Command Programs](#).

Remarks

When each block is entered, all aliases within the block are evaluated. If you alter the value of an alias at some point within a command block, commands subsequent to that point will not use the new alias value unless they are within a subordinate block.

Each block must begin with a control flow token. If you wish to create a block for the sole purpose of evaluating aliases, you should prefix it with the [.block](#) token.

`{} (Alias Interpreter)`

Article • 10/25/2023

A dollar sign followed by a pair of braces (`{}`) evaluates to a variety of values related to the specified user-named alias.

dbgcmd

```
Text ${Alias} Text
Text ${/d:Alias} Text
Text ${/f:Alias} Text
Text ${/n:Alias} Text
Text ${/v:Alias} Text
```

Parameters

Alias

Specifies the name of the alias to be expanded or evaluated. *Alias* must be a user-named alias or the *Variable* value used by the `.foreach` token.

`/d`

Evaluates to one or zero depending on whether the alias is currently defined. If the alias is defined, `$/d:Alias` is replaced by 1; if the alias is not defined, `$/d:Alias` is replaced by 0.

`/f`

Evaluates to the alias equivalent if the alias is currently defined. If the alias is defined, `$/f:Alias` is replaced by the alias equivalent; if the alias is not defined, `$/f:Alias` is replaced by an empty string.

`/n`

Evaluates to the alias name if the alias is currently defined. If the alias is defined, `$/n:Alias` is replaced by the alias name; if the alias is not defined, `$/n:Alias` is not replaced but retains its literal value of `$/n:Alias`.

`/v`

Prevents any alias evaluation. Regardless of whether *Alias* is defined, `$/v:Alias` always retains its literal value of `$/v:Alias`.

Additional Information

For an explanation of how to use aliases, see [Using Aliases](#).

Remarks

If no switches are used and the alias is currently defined, `${Alias}` is replaced by the alias equivalent. If no switches are used and the alias is not defined, `${Alias}` always retains its literal value of `${Alias}`.

One advantage of using the `${ }` token is that the alias will be evaluated even if it is adjacent to other characters. Without this token, the debugger only replaces aliases that are separated from other tokens by a space.

As indicated, there are circumstances where the `${ }` token is not replaced by anything but retains its literal value. This occurs when no switch is used and *Alias* is undefined, when the /n switch is used and *Alias* is undefined, and always when the /v switch is used. In these circumstances, the token retains its literal value, including the dollar sign and the braces. Therefore, if this is used as the parameter of a command, a syntax error will result, unless that parameter accepts arbitrary text strings.

There is, however, one exception to this. If you use `${/v:Alias}` as the first parameter to the [as \(Set Alias\)](#) or [aS \(Set Alias\)](#) command, this token will be treated as the string *Alias* alone, not as the string `${/v:Alias}`. This only works with the as, aS, and ad commands, and it only works when the /v switch is used—it will not work with `${/n:Alias}` or `${Alias}` when they retain their literal values.

Alias must be a user-named alias or the *Variable* value used by the .foreach token—not a fixed-name alias. If there is a fixed-name alias within the string *Alias*, it will be replaced before the `${ }` token is evaluated.

\$\$ (Comment Specifier)

Article • 10/25/2023

If two dollar signs (`$$`) appear at the start of a command, then the rest of the line is treated as a comment, unless the comment is terminated by a semicolon.

```
dbgcmd
```

```
$$ [any text]
```

Remarks

The `$$` token is parsed like any other debugger command. Therefore, if you want to create a comment after another command, you must precede the `$$` token with a semicolon.

The `$$` token will cause the text after it to be ignored until the end of the line or until a semicolon is encountered. A semicolon terminates the comment. Text after the semicolon is parsed as a standard command. This differs from the [* \(Comment Line Specifier\)](#) which makes the remainder of the line a comment even if a semicolon is present.

For example, the following command will display eax and ebx, but not ecx:

```
Console
```

```
0:000> r eax; $$ some text; r ebx; * more text; r ecx
```

Text prefixed by the `*` or `$$` tokens is not processed in any way. If you are performing remote debugging, a comment entered in the debugging server will not be visible in the debugging client, nor vice-versa. If you wish to make comment text appear in the Debugger Command window in a way visible to all parties, you should use [.echo \(Echo Comment\)](#).

* (Comment Line Specifier)

Article • 10/25/2023

If the asterisk (`*`) character is at the start of a command, then the rest of the line is treated as a comment, even if a semicolon appears after it.

```
dbgcmd
  * [any text]
```

Remarks

The `*` token is parsed like any other debugger command. Therefore, if you want to create a comment after another command, you must precede the `*` token with a semicolon.

The `*` token will cause the remainder of the line to be ignored, even if a semicolon appears after it. This differs from [\\$\\$ \(Comment Specifier\)](#), which creates a comment that can be terminated by a semicolon.

For example, the following command will display eax and ebx, but not ecx:

```
Console
 0:000> r eax; $$ some text; r ebx; * more text; r ecx
```

Text prefixed by the `*` or `$$` tokens is not processed in any way. If you are performing remote debugging, a comment entered in the debugging server will not be visible in the debugging client, nor vice-versa. If you wish to make comment text appear in the Debugger Command window in a way visible to all parties, you should use [.echo \(Echo Comment\)](#).

.block

Article • 10/25/2023

The **.block** token performs no action; it is used solely to introduce a block of statements.

```
dbgcmd
```

```
Commands ; .block { Commands } ; Commands
```

Additional Information

For information about using a new block to evaluate an alias, see [Using Aliases](#) and [as, aS \(Set Alias\)](#).

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

Blocks of commands are surrounded by braces. When each block is entered, all aliases within the block are evaluated. If you alter the value of an alias at some point within a command block, commands subsequent to that point will not use the new alias value unless they are within a subordinate block.

Each block must begin with a control flow token. If you wish to create a block for the sole purpose of evaluating aliases, you should prefix it with the **.block** token, since this token has no effect other than to allow a block to be introduced.

.break

Article • 10/25/2023

The **.break** token behaves like the **break** keyword in C.

```
dbgcmd  
.for (...) { ... ; .if (Condition) .break ; ...}  
.while (...) { ... ; .if (Condition) .break ; ...}  
.do { ... ; .if (Condition) .break ; ...} (...)
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

The **.break** token can be used within any **.for**, **.while**, or **.do** loop.

Since there is no control flow token equivalent to the C **goto** statement, you will usually use the **.break** token within an **.if** conditional, as shown in the syntax examples above. However, this is not actually required.

.catch

Article • 10/25/2023

The **.catch** token is used to prevent a program from terminating if an error occurs.

It does not behave like the **catch** keyword in C++.

dbgsyntax

```
Commands ; .catch { Commands } ; Commands
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

The **.catch** token is followed by braces enclosing one or more commands.

If a command within a **.catch** block generates an error, the error message is displayed, all remaining commands within the braces are ignored, and execution resumes with the first command after the closing brace.

If **.catch** is not used, an error will terminate the entire debugger command program.

You can use [.leave](#) to exit from a **.catch** block.

.continue

Article • 10/25/2023

The **.continue** token behaves like the **continue** keyword in C.

dbgsyntax

```
.for (...) { ... ; .if (Condition) .continue ; ... }  
.while (...) { ... ; .if (Condition) .continue ; ... }  
.do { ... ; .if (Condition) .continue ; ... } (...)
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

The **.continue** token can be used within any **.for**, **.while**, or **.do** loop.

Since there is no control flow token equivalent to the C `goto` statement, you will usually use the **.continue** token within an **.if** conditional, as shown in the syntax examples above. However, this is not actually required.

.do

Article • 10/25/2023

The **.do** token behaves like the **do** keyword in C, except that the word "while" is not used before the condition.

dbgcmd

```
.do { Commands } (Condition)
```

Syntax Elements

Commands

Specifies one or more commands that will be executed repeatedly as long as the condition is true -- but will always be executed at least once. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Condition

Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

The [.break](#) and [.continue](#) tokens can be used to exit or restart the *Commands* block.

.else

Article • 10/25/2023

The **.else** token behaves like the **else** keyword in C.

```
dbgcmd
```

```
.if (Condition) { Commands } .else { Commands }

.if (Condition) { Commands } .elseif (Condition) { Commands } .else {
Commands }
```

Syntax Elements

Commands

Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

.elsif

Article • 10/25/2023

The **.elsif** token behaves like the **else if** keyword combination in C.

dbgcmd

```
.if (Condition) { Commands } .elsif (Condition) { Commands }

.if (Condition) { Commands } .elsif (Condition) { Commands } .else {
Commands }
```

Syntax Elements

Condition

Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

Commands

Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

.for

Article • 10/25/2023

The **.for** token behaves like the **for** keyword in C, except that multiple increment commands must be separated by semicolons, not by commas.

dbgcmd

```
.for (InitialCommand ; Condition ; IncrementCommands) { Commands }
```

Syntax Elements

InitialCommand

Specifies a command that will be executed before the loop begins. Only a single initial command is permitted.

Condition

Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

IncrementCommands

Specifies one or more commands that will be executed at the conclusion of each loop. If you wish to use multiple increment commands, separate them by semicolons but do not enclose them in braces.

Commands

Specifies one or more commands that will be executed repeatedly as long as the condition is true. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

If all the work is being done by the increment commands, you can omit *Condition* entirely and simply use an empty pair of braces.

Here is an example of a `.for` statement with multiple increment commands:

```
dbgcmd
```

```
0:000> .for (r eax=0; @eax < 7; r eax=@eax+1; r ebx=@ebx+1) { .... }
```

The `.break` and `.continue` tokens can be used to exit or restart the *Commands* block.

.foreach

Article • 10/25/2023

The **.foreach** token parses the output of one or more debugger commands and uses each value in this output as the input to one or more additional commands.

dbgcmd

```
.foreach [Options] ( Variable { InCommands } ) { OutCommands }

.foreach [Options] /s ( Variable "InString" ) { OutCommands }

.foreach [Options] /f ( Variable "InFile" ) { OutCommands }
```

Syntax Elements

Options Can be any combination of the following options:

/pS InitialSkipNumber

Causes some initial tokens to be skipped. *InitialSkipNumber* specifies the number of output tokens that will not be passed to the specified *OutCommands*.

/ps SkipNumber

Causes tokens to be skipped repeatedly each time a command is processed. After each time a token is passed to the specified *OutCommands*, a number of tokens equal to the value of *SkipNumber* will be ignored.

Variable

Specifies a variable name. This variable will be used to hold the output from each command in the *InCommands* string; you can reference *Variable* by name in the parameters passed to the *OutCommands*. Any alphanumeric string can be used, although using a string that can also pass for a valid hexadecimal number or debugger command is not recommended. If the name used for *Variable* happens to match an existing global variable, local variable, or alias, their values will not be affected by the **.foreach** command.

InCommands

Specifies one or more commands whose output will be parsed; the resulting tokens will be passed to *OutCommands*. The output from *InCommands* is not displayed.

InString

Used with */s*. Specifies a string that will be parsed; the resulting tokens will be passed to

OutCommands.

InFile

Used with **/f**. Specifies a text file that will be parsed; the resulting tokens will be passed to *OutCommands*. The file name *InFile* must be enclosed in quotation marks.

OutCommands

Specifies one or more commands which will be executed for each token. Whenever the *Variable* string occurs it will be replaced by the current token.

Note When the string *Variable* appears within *OutCommands*, it must be surrounded by spaces. If it is adjacent to any other text -- even a parenthesis -- it will not be replaced by the current token value, unless you use the [\\${ } \(Alias Interpreter\)](#) token.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

When the output from *InCommands*, the *InString* string, or the *InFile* file is parsed, any number of spaces, tabs, or carriage returns is treated as a single delimiter. Each of the resulting pieces of text is used to replace *Variable* when it appears within *OutCommands*.

Here is an example of a **.foreach** statement that uses the [dds](#) command on each token found in the file *myfile.txt*:

```
dbgcmd  
0:000> .foreach /f ( place "g:\myfile.txt") { dds place }
```

The **/pS** and **/ps** flags can be used to pass only certain tokens to the specified *OutCommands*. For example, the following statement will skip the first two tokens in the *myfile.txt* file and then pass the third to [dds](#). After each token that is passed, it will skip four tokens. The result is that [dds](#) will be used with the 3rd, 8th, 13th, 18th, and 23rd tokens, and so on:

```
dbgcmd  
0:000> .foreach /pS 2 /ps 4 /f ( place "g:\myfile.txt") { dds place }
```

For more examples that use the `.foreach` token, see [Debugger Command Program Examples](#).

.if

Article • 10/25/2023

The `.if` token behaves like the `if` keyword in C.

```
dbgcmd

.if (Condition) { Commands }

.if (Condition) { Commands } .else { Commands }

.if (Condition) { Commands } .elseif (Condition) { Commands }

.if (Condition) { Commands } .elseif (Condition) { Commands } .else {
Commands }
```

Syntax Elements

Condition

Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

Commands

Specifies one or more commands that will be executed conditionally. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

.leave

Article • 10/25/2023

The **.leave** token is used to exit from a [.catch](#) block.

```
dbgcmd
```

```
.catch { ... ; .if (Condition) .leave ; ... }
```

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

When a **.leave** token is encountered within a [.catch](#) block, the program exits from the block, and execution resumes with the first command after the closing brace.

Since there is no control flow token equivalent to the C **goto** statement, you will usually use the **.leave** token within an [.if](#) conditional, as shown in the syntax examples above. However, this is not actually required.

.printf

Article • 10/25/2023

The **.printf** token behaves like the **printf** statement in C.

dbgcmd

```
.printf [/D] [Option] "FormatString" [, Argument , ...]
```

Syntax Elements

/D

Specifies that the format string contains [Debugger Markup Language](#) (DML).

Option

(WinDbg only) Specifies the type of text message that WinDbg should interpret the FormatString as. WinDbg assigns each type of Debugger Command window message a background and text color; choosing one of these options causes the message to be displayed in the appropriate colors. The default is to display the text as a normal-level message.

The following options are available.

Option	Type of message	Title of colors in Options dialog box
/od	debuggee	Debuggee level command window
/oD	debuggee prompt	Debuggee prompt level command window
/oe	error	Error level command window
/on	normal	Normal level command window
/op	prompt	Prompt level command window
/oP	prompt registers	Prompt registers level command window
/os	symbols	Symbol message level command window

Option	Type of message	Title of colors in Options dialog box
/ov	verbose	Verbose level command window
/ow	warning	Warning level command window

FormatString

Specifies the format string, as in `printf`. In general, conversion characters work exactly as in C. For the floating-point conversion characters, the 64-bit argument is interpreted as a 32-bit floating-point number unless the I modifier is used.

The "I64" modifier can be added to indicate that a value should be interpreted as 64-bits. For instance, "%I64x" can be used to print a 64-bit hexadecimal number.

The %p conversion character is supported, but it represents a pointer in the target's virtual address space. It must not have any modifiers and it uses the debugger's internal address formatting. In addition to the standard printf-style format specifiers, the following additional conversion characters are supported.

Character	Argument type	Argument	Text printed
%p	ULONG64	A pointer in the target's virtual address space.	The value of the pointer.
%N	DWORD_PTR (32 or 64 bits, depending on the host's architecture)	A pointer in the host's virtual address space.	The value of the pointer. (This is equivalent to the standard C %p character.)
%ma	ULONG64	The address of a NULL-terminated ASCII string in the target's virtual address space.	The specified string.
%mu	ULONG64	The address of a NULL-terminated Unicode string in the target's virtual address space.	The specified string.
%msa	ULONG64	The address of an ANSI_STRING	The specified string.

Character	Argument type	Argument	Text printed
		structure in the target's virtual address space.	
%msu	ULONG64	The address of a UNICODE_STRING structure in the target's virtual address space.	The specified string.
%y	ULONG64	The address of a debugger symbol in the target's virtual address space.	A string containing the name of the specified symbol (and displacement, if any).
%ly	ULONG64	The address of a debugger symbol in the target's virtual address space.	A string containing the name of the specified symbol (and displacement, if any), as well as any available source line information.

Arguments

Specifies arguments for the format string, as in `printf`. The number of arguments that are specified should match the number of conversion characters in *FormatString*. Each argument is an expression that will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

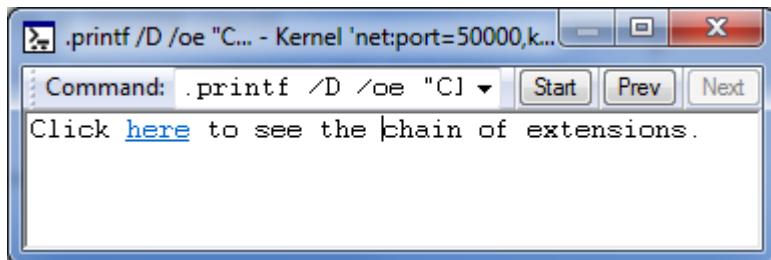
Remarks

The color settings that you can choose by using the *Options* parameter are by default all set to black text on a white background. To make best use of these options, you must first use **View | Options** to open the Options dialog box and change the color settings for Debugger Command window messages.

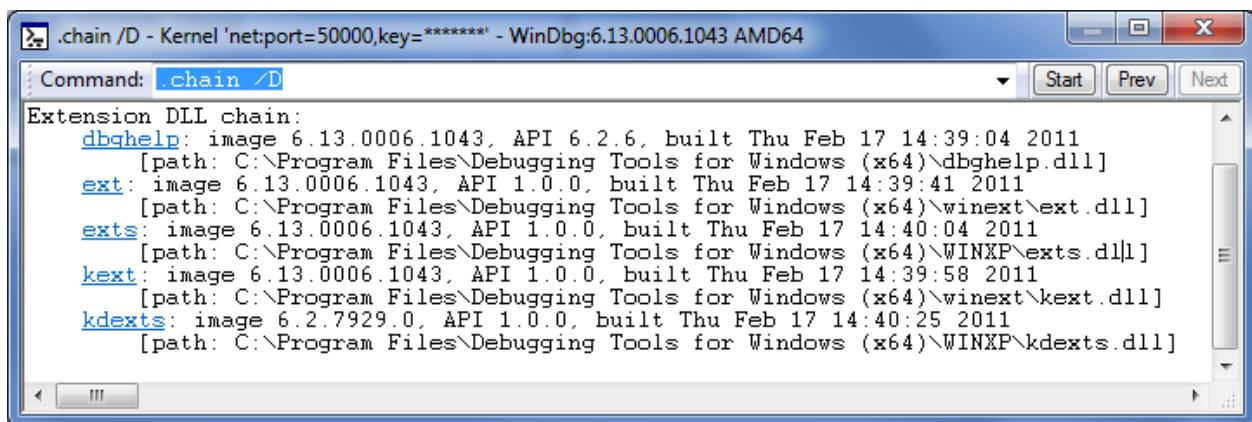
The following example shows how to include a DML tag in the format string.

```
dbgcmd
```

```
.printf /D "Click <link cmd=\".chain /D\">here</link> to see extensions  
DLLs."
```



The output shown in the preceding image has a link that you can click to execute the command specified in the `<link>` tag. The following image shows the result of clicking the link.



For information about DML tags, see `dml.doc` in the installation folder for Debugging Tools for Windows.

.while

Article • 10/25/2023

The **.while** token behaves like the **while** keyword in C.

dbgcmd

```
.while (Condition) { Commands }
```

Syntax Elements

Condition

Specifies a condition. If this evaluates to zero, it is treated as false; otherwise it is true. Enclosing *Condition* in parentheses is optional. *Condition* must be an expression, not a debugger command. It will be evaluated by the default expression evaluator (MASM or C++). For details, see [Numerical Expression Syntax](#).

Commands

Specifies one or more commands that will be executed repeatedly as long as the condition is true. This block of commands needs to be enclosed in braces, even if it consists of a single command. Multiple commands should be separated by semicolons, but the final command before the closing brace does not need to be followed by a semicolon.

Additional Information

For information about other control flow tokens and their use in debugger command programs, see [Using Debugger Command Programs](#).

Remarks

The [.break](#) and [.continue](#) tokens can be used to exit or restart the *Commands* block.

Commands

Article • 10/25/2023

This section of the reference discusses the various debugger commands that you can use in CDB, KD, and WinDbg.

- [ENTER \(Repeat Last Command\)](#)
- [\\$<, \\$><, \\$\\$<, \\$\\$><, \\$\\$>a< \(Run Script File\)](#)
- [? \(Command Help\)](#)
- [? \(Evaluate Expression\)](#)
- [?? \(Evaluate C++ Expression\)](#)
- [# \(Search for Disassembly Pattern\)](#)
- [|| \(System Status\)](#)
- [||s \(Set Current System\)](#)
- [| \(Process Status\)](#)
- [|s \(Set Current Process\)](#)
- [~ \(Thread Status\)](#)
- [~e \(Thread-Specific Command\)](#)
- [~f \(Freeze Thread\)](#)
- [~u \(Unfreeze Thread\)](#)
- [~n \(Suspend Thread\)](#)
- [~m \(Resume Thread\)](#)
- [~s \(Set Current Thread\)](#)
- [~s \(Change Current Processor\)](#)
- [a \(Assemble\)](#)
- [ad \(Delete Alias\)](#)
- [ah \(Assertion Handling\)](#)
- [al \(List Aliases\)](#)
- [as, aS \(Set Alias\)](#)
- [ba \(Break on Access\)](#)
- [bc \(Breakpoint Clear\)](#)
- [bd \(Breakpoint Disable\)](#)
- [be \(Breakpoint Enable\)](#)
- [bl \(Breakpoint List\)](#)
- [bp, bu, bm \(Set Breakpoint\)](#)
- [br \(Breakpoint Rerumber\)](#)
- [bs \(Update Breakpoint Command\)](#)
- [bsc \(Update Conditional Breakpoint\)](#)
- [c \(Compare Memory\)](#)
- [d, da, db, dc, dd, dD, df, dp, dq, du, dw, dW, dyb, dyd \(Display Memory\)](#)

- dda, ddp, ddu, dpa, dpp, dpu, dqa, dqp, dqu (Display Referenced Memory)
- dds, dps, dqs (Display Words and Symbols)
- dg (Display Selector)
- dl (Display Linked List)
- ds, dS (Display String)
- dt (Display Type)
- dtx (Display Type - Extended Debugger Object Model Information)
- dv (Display Local Variables)
- dx (Display Debugger Object Model Expression)
- e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza, ezu (Enter Values)
- f, fp (Fill Memory)
- g (Go)
- gc (Go from Conditional Breakpoint)
- gh (Go with Exception Handled)
- gn, gN (Go with Exception Not Handled)
- gu (Go Up)
- ib, iw, id (Input from Port)
- j (Execute If - Else)
- k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)
- l+, l- (Set Source Options)
- ld (Load Symbols)
- lm (List Loaded Modules)
- ln (List Nearest Symbols)
- ls, lsa (List Source Lines)
- lsc (List Current Source)
- lse (Launch Source Editor)
- lsf, lsf- (Load or Unload Source File)
- lsp (Set Number of Source Lines)
- m (Move Memory)
- n (Set Number Base)
- ob, ow, od (Output to Port)
- p (Step)
- pa (Step to Address)
- pc (Step to Next Call)
- pct (Step to Next Call or Return)
- ph (Step to Next Branching Instruction)
- pt (Step to Next Return)
- q, qq (Quit)
- qd (Quit and Detach)
- r (Registers)

- rdmsr (Read MSR)
- rm (Register Mask)
- s (Search Memory)
- so (Set Kernel Debugging Options)
- sq (Set Quiet Mode)
- ss (Set Symbol Suffix)
- sx, sxd, sxe, sxi, sxn, sxr, sx- (Set Exceptions)
- t (Trace)
- ta (Trace to Address)
- tb (Trace to Next Branch)
- tc (Trace to Next Call)
- tct (Trace to Next Call or Return)
- th (Trace to Next Branching Instruction)
- tt (Trace to Next Return)
- u (Unassemble)
- uf (Unassemble Function)
- up (Unassemble from Physical Memory)
- ur (Unassemble Real Mode BIOS)
- ux (Unassemble x86 BIOS)
- vercommand (Show Debugger Command Line)
- version (Show Debugger Version)
- vertarget (Show Target Computer Version)
- wrmsr (Write MSR)
- wt (Trace and Watch Data)
- x (Examine Symbols)
- z (Execute While)

ENTER (Repeat Last Command)

Article • 11/14/2023

The ENTER key repeats the last command that you typed.



Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In CDB and KD, pressing the ENTER key by itself at a command prompt reissues the command that you previously entered.

In WinDbg, the ENTER key can have no effect or you can use it to repeat the previous command. You can set this option in the **Options** dialog box. (To open the Options dialog box, click **Options** on the **View** menu or click the **Options** button ().

If you set ENTER to repeat the last command, but you want to create white space in the [Debugger Command window](#), use the *** (Comment Line Specifier)** token and then press ENTER several times.

\$<, \$><, \$\$<, \$\$><, \$\$>a< (Run Script File)

Article • 10/25/2023

The \$<, \$><, \$\$<, \$\$><, and \$\$>a< commands read the contents of the specified script file and use its contents as debugger command input.

dbgcmd

```
$<Filename  
$><Filename  
$$<Filename  
$$><Filename  
$$>a<Filename [arg1 arg2 arg3 ...]
```

Parameters

Filename Specifies a file that contains valid debugger command text. The file name must follow Microsoft Windows file name conventions. The file name may contain spaces.

argn

Specifies any number of string arguments for the debugger to pass to the script. The debugger will replace any string of the form \${\$argn} in the script file with the corresponding *argn* before executing the script. Arguments may not contain quotation marks or semicolons. Multiple arguments must be separated by spaces; if an argument contains a space it must be enclosed in quotation marks. All arguments are optional.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The `$$<` and `$<` tokens execute the commands that are found in the script file literally. However, with `$<` you can specify any file name, including one that contains semicolons. Because `$<` allows semicolons to be used in the file name, you cannot concatenate `$<` with other debugger commands, because a semicolon cannot be used both as a command separator and as part of a file name.

The `$$><` and `$><` tokens execute the commands that are found in the script file literally, which means they open the script file, replace all carriage returns with semicolons, and execute the resulting text as a single command block. As with `$<` discussed previously, the `$><` variation permits file names that contain semicolons, which means you cannot concatenate `$><` with other debugger commands.

The `$$><` and `$><` tokens are useful if you are running scripts that contain debugger command programs. For more information about these programs, see [Using Debugger Command Programs](#).

Unless you have file names that contain semicolons, you do not need to use either `$<` or `$><`.

The `$$>a<` token allows the debugger to pass arguments to the script. If *Filename* contains spaces, it must be enclosed in quotation marks. If too many arguments are supplied, the excess arguments are ignored. If too few arguments are supplied, any token in the source file of the form `$$argn` where *n* is larger than the number of supplied arguments will remain in its literal form and will not be replaced with anything. You can follow this command with a semicolon and additional commands; the presence of a semicolon terminates the argument list.

When the debugger executes a script file, the commands and their output are displayed in the [Debugger Command window](#). When the end of the script file is reached, control returns to the debugger.

The following table summarizes how you can use these tokens.

Token	Allows file names that contain semicolons	Allows concatenation of additional commands separated by semicolons	Condenses to single command block	Allows script arguments
<code>\$<</code>	Yes	No	No	No
<code>\$><</code>	Yes	No	Yes	No
<code>\$\$<</code>	No	Yes	No	No

Token	Allows file names that contain semicolons	Allows concatenation of additional commands separated by semicolons	Condenses to single command block	Allows script arguments
\$\$><	No	Yes	Yes	No
\$\$>a<	No	Yes	Yes	Yes

The `$<`, `$><`, `$$<`, and `$$><` commands echo the commands contained in the script file and display the output of these commands. The `$$>a<` command does not echo the commands found in the script file, but merely displays their output.

Script files can be nested. If the debugger encounters one of these tokens in a script file, execution moves to the new script file and returns to the previous location when the new script file has been completed. Scripts can also be called recursively.

In WinDbg, you can paste the additional command text in the Debugger Command window.

Examples

The following example demonstrates how to pass arguments to a script file, `Myfile.txt`. Assume that the file contains the following text:

```
Console

.echo The first argument is ${$arg1}.
.echo The second argument is ${$arg2}.
```

Then you can pass arguments to this file by using a command like this:

```
Console

0:000> $$>a<myfile.txt myFirstArg mySecondArg
```

The result of this command would be:

```
Console

The first argument is myFirstArg.
The second argument is mySecondArg.
```

Here is an example of what happens when the wrong number of argument is supplied. Assume that the file My Script.txt contains the following text:

Console

```
.echo The first argument is ${arg1}.\n.echo The fifth argument is ${arg5}.\n.echo The fourth argument is ${arg4}.
```

Then the following semicolon-delimited command line produces output thus:

Console

```
0:000> $$>a< "c:\bin1\my script.txt" "First one" Two "Three More" Four; recx\nThe first argument is First one.\nThe fifth argument is ${arg5}.\nThe fourth argument is Four.\necx=0021f4ac
```

In the preceding example, the file name is enclosed in quotation marks because it contains a space, and arguments that contain spaces are enclosed in quotation marks as well. Although a fifth argument seems to be expected by the script, the semicolon terminates the \$\$>a< command after the fourth argument.

? (Command Help)

Article • 10/25/2023

The question mark (?) character displays a list of all commands and operators.

Note A question mark by itself displays command help. The [? expression](#) syntax evaluates the given expression.

dbgcmd
?

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

For more information about standard commands, use ?. For more information about meta-commands, use [.help](#). For more information about extension commands, use [!help](#).

? (Evaluate Expression)

Article • 10/25/2023

The question mark (?) command evaluates and displays the value of an expression.

A question mark by itself (?) displays command help. The ? *expression* command evaluates the given expression.

dbgcmd
? Expression

Parameters

Expression

Specifies the expression to evaluate.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The input and output of the ? command depend on whether you are using MASM expression syntax or C++ expression syntax. For more information about these kinds of expression syntax, see [Evaluating Expressions](#) and [Numerical Expression Syntax](#).

If you are using MASM syntax, the input and output depend on the current radix. To change the radix, use the [n \(Set Number Base\)](#) command.

The ? command evaluates symbols in the expression in the context of the current thread and process.

Some strings may contain escapes, such as `\n`, `\"`, `\r`, and `\b`, that are meant to be read literally, rather than interpreted by the evaluator. If an escape within a string is interpreted by the evaluator, errors in evaluation can occur. For example:

```
Console

0:000> as AliasName c:\dir\name.txt
0:000> al
    Alias          Value
    -----
AliasName      c:\dir\name.txt
0:001> ? $spat( "c:\dir\name.txt", "*name*" )
Evaluate expression: 0 = 00000000

0:001> ? $spat( "${AliasName}", "*name*" )
Evaluate expression: 0 = 00000000

0:001> ? $spat( "c:\dir\", "*filename*" )
Syntax error at '( "c:\dir\", "*filename*" )'
```

In the first two examples, even though the string does match the pattern, the evaluator is returning a value of FALSE. In the third, the evaluator cannot make a comparison because the string ends in a backslash (`\`), and so the `\"` is translated by the evaluator.

To get the evaluator to interpret a string literally, you must use the `@"String"` syntax. The following code example shows the correct results:

```
Console

0:000> ? $spat( @"c:\dir\name.txt", "*name*" )
Evaluate expression: 1 = 0000000`00000001

0:000> ? $spat( @"${AliasName}", "*name*" )
Evaluate expression: 1 = 0000000`00000001

0:001> ? $spat( @"c:\dir\", "*filename*" )
Evaluate expression: 0 = 00000000
```

In the preceding examples, the `$spat` MASM operator checks the first string to determine whether it matches (case-insensitive) the pattern of the second string. For more information about MASM operators, see the [MASM Numbers and Operators](#) topic.

See also

[?? \(Evaluate C++ Expression\)](#)

.formats (Show Number Formats)

MASM Numbers and Operators

C++ Numbers and Operators

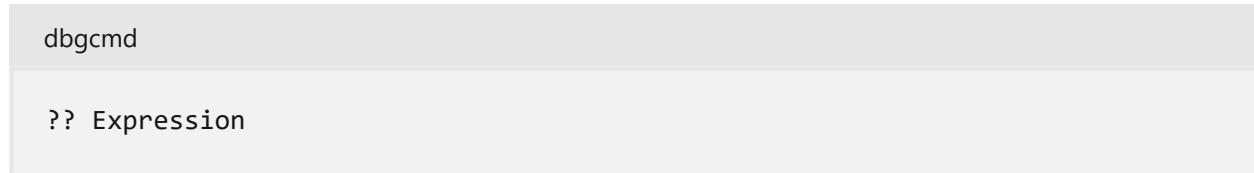
MASM Expressions vs. C++ Expressions

Mixed Expression Examples

?? (Evaluate C++ Expression)

Article • 10/25/2023

The double question mark (??) command evaluates and displays the value of an expression according to C++ expression rules.



Parameters

Expression

Specifies the C++ expression to evaluate. For more information about the syntax, see [C++ Numbers and Operators](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The ?? command evaluates symbols in the expression in the context of the current thread and process.

If you want to evaluate a part of the expression according to MASM expression rules, enclose that part in parentheses and add two at signs (@@) before it. For more information about MASM expressions and C++ expressions, see [Evaluating Expressions](#) and [Numerical Expression Syntax](#).

See also

[? \(Evaluate Expression\)](#)

.formats (Show Number Formats)

Evaluating Expressions

Numerical Expression Syntax

(Search for Disassembly Pattern)

Article • 10/25/2023

The number sign (#) command searches for the specified pattern in the disassembly code.

dbgcmd

```
# [Pattern] [Address [ L Size ]]
```

Parameters

Pattern

Specifies the pattern to search for in the disassembly code. *Pattern* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#). If you want to include spaces in *Pattern*, you must enclose the pattern in quotation marks. The pattern is not case sensitive. If you have previously used the # command and you omit *Pattern*, the command reuses the most recently used pattern.

Address

Specifies the address where the search begins. For more information about the syntax, see [Address and Address Range Syntax](#).

Size

Specifies the number of instructions to search. If you omit *Size*, the search continues until the first match occurs.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

If you previously used the # command and you omit *Address*, the search begins where the previous search ended.

This command works by searching the disassembled text for the specified pattern. You can use this command to find register names, constants, or any other string that appears in the disassembly output. You can repeat the command without the *Address* parameter to find successive occurrences of the pattern.

You can view disassembly instructions by using the [u \(Unassemble\)](#) command or by using the [Disassembly window](#) in WinDbg. The disassembly display contains up to four parts: Address offset, Binary code, Assembly language mnemonic, and Assembly language details. The following example shows a possible display.

```
dbgcmd

0040116b    45          inc      ebp
0040116c    fc          cld
0040116d    8945b0     mov      eax,[ebp-0x1c]
```

The # command can search for text within any single part of the disassembly display. For example, you could use # eax 0040116b to find the `mov eax,[ebp-0x1c]` instruction at address 0040116d. The following commands also find this instruction.

```
dbgcmd

# [ebp?0x 0040116b
# mov 0040116b
# 8945* 0040116b
# 116d 0040116b
```

However, you cannot search for `mov eax*` as a single unit, because mov and eax appear in different parts of the display. Instead, use `mov*eax`.

As an additional example, you could issue the following command to search for the first reference to the `strlen` function after the entry point `main`.

```
dbgcmd
```

```
# strlen main
```

Similarly, you could issue the following two commands to find the first jnz instruction after address 0x779F9FBA and then find the next jnz instruction after that.

```
dbgcmd
```

```
# jnz 779f9fba#
```

When you omit *Pattern* or *Address*, their values are based on the previous use of the # command. If you omit either parameter the first time that you issue the # command, no search is performed. However, the values of *Pattern* and *Address* are initialized even in this situation.

If you include *Pattern* or *Address*, its value is set to the entered value. If you omit *Address*, it is initialized to the current value of the program counter. If you omit *Pattern*, it is initialized to an empty pattern.

|| (System Status)

Article • 10/25/2023

The double vertical bar (||) command prints status for the specified system or for all systems that you are currently debugging.

Do not confuse this command with the [| \(Process Status\)](#) command.

```
dbgcmd
```

```
|| System
```

Parameters

System

Specifies the system to display. If you omit this parameter, all systems that you are debugging are displayed. For more information about the syntax, see [System Syntax](#).

Environment

Item	Description
Modes	Multiple target debugging
Targets	Live, crash dump
Platforms	All

Remarks

The || command is useful only when you are debugging multiple targets. Many, but not all, multiple-target debugging sessions involve multiple systems. For more information about these sessions, see [Debugging Multiple Targets](#).

Each system listing includes the server name and the protocol details. The system that the debugger is running on is identified as <Local>.

The following examples show you how to use this command. The following command displays all systems.

```
dbgcmd
```

```
3:2:005> ||
```

The following command also displays all systems.

```
dbgcmd
```

```
3:2:005> ||*
```

The following command displays the currently active system.

```
dbgcmd
```

```
3:2:005> ||.
```

The following command displays the system that had the most recent exception or break.

```
dbgcmd
```

```
3:2:005> ||#
```

The following command displays system number 2.

```
dbgcmd
```

```
3:2:005> ||2
```

||s (Set Current System)

Article • 10/25/2023

The ||s command sets or displays the current system number.

```
dbgcmd  
||System s  
|| s
```

Do not confuse this command with the [s \(Search Memory\)](#), [~s \(Change Current Processor\)](#), [~s \(Set Current Thread\)](#), or [|s \(Set Current Process\)](#) command.

Parameters

System

Specifies the system number to activate. For more information about the syntax, see [System Syntax](#).

Environment

Item	Description
Modes	Multiple target debugging
Targets	Live, crash dump
Platforms	All

Remarks

The ||s command is useful when you are debugging multiple targets or working with multiple dump files. For more information about these kinds of sessions, see [Debugging Multiple Targets](#).

If you use the ||s syntax, the debugger displays information about the current system.

This command also disassembles the current instruction for the current system, process, and thread.

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

| (Process Status)

Article • 10/25/2023

The pipe (|) command displays status for the specified process, or for all processes that you are currently debugging.

Do not confuse this command with the [|| \(System Status\)](#) command.

dbgcmd
Process

Parameters

Process

Specifies the process to display. If you omit this parameter, all processes that you are debugging are displayed. For more information about the syntax, see [Process Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see [Controlling Processes and Threads](#).

Remarks

You can specify processes only in user mode.

You can add a process symbol before many commands. For more information about the meaning of a pipe (|) followed by a command, see the entry for the command itself.

Unless you enabled the debugging of child processes when you started the debugging session, there is only one process that is available to the debugger.

The following examples show you how to use this command. The following command displays all processes.

```
dbgcmd
```

```
2:005> |
```

The following command also displays all processes.

```
dbgcmd
```

```
2:005> |*
```

The following command displays the currently active process.

```
dbgcmd
```

```
2:005> |.
```

The following command displays the process that originally caused the exception (or that the debugger originally attached to).

```
dbgcmd
```

```
2:005> |#
```

The following command displays process number 2.

```
dbgcmd
```

```
2:005> |2
```

The previous command displays the following output.

```
dbgcmd
```

```
0:002> |
# 0 id: 224    name: myprog.exe
  1 id: 228    name: onechild.exe
. 2 id: 22c    name: anotherchild.exe
```

On the first line of this output, 0 is the decimal process number, 224 is the hexadecimal process ID, and *Myprog.exe* is the application name of the process. The period (.) before process 2 means that this process is the current process. The number sign (#) before process 0 means that this process was the one that originally caused the exception or that the debugger attached to.

|s (Set Current Process)

Article • 10/25/2023

The |s command sets or displays the current process number.

Do not confuse this command with the [s \(Search Memory\)](#), [~s \(Change Current Processor\)](#), [~s \(Set Current Thread\)](#), or [||s \(Set Current System\)](#) command.

```
dbgcmd
```

```
|Process s  
| s
```

Parameters

Process

Specifies the process to set or display. For more information about the syntax, see [Process Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about other methods of displaying or controlling processes and threads, see [Controlling Processes and Threads](#).

Remarks

You can specify processes only in user mode.

If you use the |s syntax, the debugger displays information about the current process.

This command also disassembles the current instruction for the current system, process, and thread.

~ (Thread Status)

Article • 10/25/2023

The tilde (~) command displays status for the specified thread or for all threads in the current process.

dbgcmd
~ Thread

Parameters

Thread

Specifies the thread to display. If you omit this parameter, all threads are displayed. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

You can add a thread symbol before many commands. For more information about the meaning of a tilde (~) followed by a command, see the entry for the command itself.

The following examples show you how to use this command. The following command displays all threads.

```
dbgcmd
```

```
0:001> ~
```

The following command also displays all threads.

```
dbgcmd
```

```
0:001> ~*
```

The following command displays the currently active thread.

```
dbgcmd
```

```
0:001> ~.
```

The following command displays the thread that originally caused the exception (or that was active when the debugger attached to the process).

```
dbgcmd
```

```
0:001> ~#
```

The following command displays thread number 2.

```
dbgcmd
```

```
0:001> ~2
```

The previous command displays the following output.

```
dbgcmd
```

```
0:001> ~
  0 id: 4dc.470 Suspend: 0 Teb 7ffde000 Unfrozen
  . 1 id: 4dc.534 Suspend: 0 Teb 7ffdd000 Unfrozen
#  2 id: 4dc.5a8 Suspend: 0 Teb 7ffdcc000 Unfrozen
```

On the first line of this output, 0 is the decimal thread number, 4DC is the hexadecimal process ID, 470 is the hexadecimal thread ID, 0x7FFDE000 is the address of the TEB, and **Unfrozen** is the thread status. The period (.) before thread 1 means this thread is the

current thread. The number sign (#) before thread 2 means this thread was the one that originally caused the exception or it was active when the debugger attached to the process.

~e (Thread-Specific Command)

Article • 10/25/2023

The **~e** command executes one or more commands for a specific thread or for all threads in the target process.

Do not confuse this command with the [e \(Enter Values\)](#) command.

```
dbgcmd
```

```
~Thread e CommandString
```

Parameters

Thread

Specifies the thread or threads that the debugger will execute *CommandString* for. For more information about the syntax, see [Thread Syntax](#).

CommandString

Specifies one or more commands to execute. You should separate multiple commands by using semicolons. *CommandString* includes the rest of the input line. All of the text that follows the letter "e" is interpreted as part of this string. Do not enclose *CommandString* in quotation marks.

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about other commands that control threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

When you use the ~e command together with one thread, the ~e command only saves some typing. For example, the following two commands are equivalent.

```
dbgcmd  
0:000> ~2e r; k; kd  
0:000> ~2r; ~2k; ~2kd
```

However, you can use the ~e qualifier to repeat a command or extension command several times. When you use the qualifier in this manner, it can eliminate extra typing. For example, the following command repeats the [!gle](#) extension command for every thread that you are debugging.

```
dbgcmd  
0:000> ~*e !gle
```

If an error occurs in the execution of one command, execution continues with the next command.

You cannot use the ~e qualifier together with execution commands ([g](#), [gh](#), [gn](#), [gN](#), [gu](#), [p](#), [pa](#), [pc](#), [t](#), [ta](#), [tb](#), [tc](#), [wt](#)).

You cannot use the ~e qualifier together with the [j \(Execute If-Else\)](#) or [z \(Execute While\)](#) conditional commands.

If you are debugging more than one process, you cannot use the ~e command to access the virtual memory space for an inactive process.

~f (Freeze Thread)

Article • 10/25/2023

The **~f** command freezes the given thread, causing it to stop and wait until it is unfrozen.

Do not confuse this command with the [f \(Fill Memory\)](#) command.

```
dbgcmd
```

```
~Thread f
```

Parameters

Thread

Specifies the thread to freeze. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how frozen threads behave and a list of other commands that control the freezing and suspending of threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

The **~f** command causes the specified thread to freeze. When the debugger enables the target application to resume execution, other threads execute as expected while this thread remains stopped.

The following examples show you how to use this command. The following command displays the current status of all threads.

```
dbgcmd  
0:000> ~* k
```

The following command freezes the thread that caused the current exception.

```
dbgcmd  
0:000> ~# f
```

The following command checks that the status of this thread is suspended.

```
dbgcmd  
0:000> ~* k
```

The following command unfreezes thread number 123.

```
dbgcmd  
0:000> ~123 u
```

~u (Unfreeze Thread)

Article • 10/25/2023

The **~u** command unfreezes the specified thread.

Do not confuse this command with the [U \(Unassemble\) command](#).

dbgcmd
~Thread u

Parameters

Thread

Specifies the thread or threads to unfreeze. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how frozen threads behave and a list of other commands that control the freezing and suspending of threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

The following examples show you how to use the ~ commands.

The following command displays the current status of all threads.

```
dbgcmd
```

```
0:000> ~* k
```

The following command freeze the thread that caused the current exception.

```
dbgcmd
```

```
0:000> ~# f
```

The following command checks that the status of this thread is suspended.

```
dbgcmd
```

```
0:000> ~* k
```

The following command unfreezes thread number 123.

```
dbgcmd
```

```
0:000> ~123 u
```

~n (Suspend Thread)

Article • 10/25/2023

The **~n** command suspends execution of the specified thread.

Do not confuse this command with the [n \(Set Number Base\)](#) command.

dbgcmd
~Thread n

Parameters

Thread

Specifies the thread or threads to suspend. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For more information about the suspend count and how suspended threads behave and for a list of other commands that control the suspending and freezing of threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

Every time that you use the **~n** command, the thread's suspend count is increased by one.

The thread's start address is displayed when you use this command.

~m (Resume Thread)

Article • 10/25/2023

The **~m** command resumes execution of the specified thread.

Do not confuse this command with the [m \(Move Memory\)](#) command.

dbgcmd
~Thread m

Parameters

Thread

Specifies the thread or threads to resume. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For more information about the suspend count and how suspended threads behave and for a list of other commands that control the suspending and freezing of threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

Every time that you use the **~m** command, the thread's suspend count is decreased by one.

~s (Set Current Thread)

Article • 10/25/2023

The `~s` command sets or displays the current thread number in user mode.

Do not confuse this command with the [~s \(Change Current Processor\)](#) command (which works only in kernel mode), the [|s \(Set Current Process\)](#) command, the [||s \(Set Current System\)](#) command, or the [s \(Search Memory\)](#) command.

dbgcmd
<code>~Thread s</code>
<code>~ s</code>

Parameters

Thread

Specifies the thread to set or display. For more information about the syntax, see [Thread Syntax](#).

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information and other methods of displaying or controlling processes and threads, see [Controlling Processes and Threads](#).

Remarks

You can specify threads only in user mode. In kernel mode, the tilde (~) refers to a processor.

If you use the `~s` syntax, the debugger displays information about the current thread.

This command also disassembles the current instruction for the current system, process, and thread.

~s (Change Current Processor)

Article • 04/03/2024

The `~s` command sets which processor is debugged on a multiprocessor system.

In kernel mode, `~s` changes the current processor. Do not confuse this command with the [~s \(Set Current Thread\)](#) command (which works only in user mode), the [|s \(Set Current Process\)](#) command, the [||s \(Set Current System\)](#) command, or the [s \(Search Memory\)](#) command.

```
dbgcmd
```

```
~Processor s
```

Parameters

Processor

Specifies the number of the processor to debug.

Environment

[\[\] Expand table](#)

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Remarks

You can specify processors only in kernel mode. In user mode, the tilde (~) refers to a thread.

You can immediately tell when you are working on a multiple processor system by the shape of the kernel debugging prompt. In the following example, 0: means that you are debugging the first processor in the computer.

```
dbgcmd
```

```
0: kd>
```

Use the following command to switch between processors:

```
dbgcmd
```

```
0: kd> ~1s  
1: kd>
```

Now the second processor in the computer that is being debugged.

See also

[Multiprocessor Syntax](#)

Feedback

Was this page helpful?

 Yes

 No

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

a (Assemble)

Article • 10/25/2023

The **a** command assembles 32-bit x86 instruction mnemonics and puts the resulting instruction codes into memory.

dbgcmd

a [Address]

Parameters

Address

Specifies the beginning of the block in memory where the resulting codes are put. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

The **a** command does not support 64-bit instruction mnemonics. However, the **a** command is enabled regardless of whether you are debugging a 32-bit target or a 64-bit target. Because of the similarities between x86 and x64 instructions, you can sometimes use the **a** command successfully when debugging a 64-bit target.

If you do not specify an address, the assembly starts at the address that the current value of the instruction pointer specifies. To assemble a new instruction, type the desired

mnemonic and press ENTER. To end assembly, press only ENTER.

Because the assembler searches for all of the symbols that are referred to in the code, this command might take time to complete. During this time, you cannot press **CTRL+C** to end the **a** command.

ad (Delete Alias)

Article • 10/25/2023

The **ad** command deletes an alias from the alias list.

```
dbgcmd  
ad [/q] Name  
ad *
```

Parameters

/q

Specifies quiet mode. This mode hides the error message if the alias that *Name* specifies does not exist.

Name

Specifies the name of the alias to delete. If you specify an asterisk (*), all aliases are deleted (even if there is an alias whose name is "*").

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see [Using Aliases](#).

Remarks

You can use the **ad** command to delete any user-named alias. But you cannot use this command to delete a fixed-name alias (\$u0 to \$u9).

ah (Assertion Handling)

Article • 10/25/2023

The **ah** command controls the assertion handling status for specific addresses.

```
dbgcmd  
  ahb [Address]  
  ahi [Address]  
  ahd [Address]  
  ahc  
  ah
```

Parameters

ahb

Breaks into the debugger if an assertion fails at the specified address.

ahi

Ignores an assertion failure at the specified address.

ahd

Deletes any assertion-handling information at the specified address. This deletion causes the debugger to return to its default state for that address.

Address

Specifies the address of the instruction whose assertion-handling status is being set. If you omit this parameter, the debugger uses the current program counter.

ahc

Deletes all assertion-handling information for the current process.

ah

Displays the current assertion-handling settings.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about break status and handling status, descriptions of all event codes, a list of the default status for all events, and details about other methods of controlling this status, see [Controlling Exceptions and Events](#).

Remarks

The **ah*** command controls the assertion handling status for a specific address. The [**sx***](#) **asrt** command controls the global assertion handling status. If you use **ah*** for a certain address and then an assert occurs there, the debugger responds based on the **ah*** settings and ignores the [**sx***](#) **asrt** settings.

When the debugger encounters an assertion, the debugger first checks whether handling has been configured for that specific address. If you have not configured the handling, the debugger uses the global setting.

The **ah*** command affects only the current process. When the current process ends, all status settings are lost.

Assertion handling status affects only STATUS_ASSERTION_EXCEPTION exceptions. This handling does not affect the kernel-mode ASSERT routine.

al (List Aliases)

Article • 10/25/2023

The **al** command displays a list of all currently defined user-named aliases.

```
dbgcmd
  al
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see [Using Aliases](#).

Remarks

The **al** command lists all user-named aliases. But this command does not list fixed-name aliases (\$u0 to \$u9).

as, aS (Set Alias)

Article • 10/25/2023

The **as** and **aS** commands define a new alias or redefine an existing one.

```
dbgcmd

as Name EquivalentLine
aS Name EquivalentPhrase
aS Name "EquivalentPhrase"
as /e Name EnvironmentVariable
as /ma Name Address
as /mu Name Address
as /msa Name Address
as /msu Name Address
as /x Name Expression
aS /f Name File
as /c Name CommandString
```

Parameters

Name

Specifies the alias name. This name can be any text string that does not contain a space or the ENTER keystroke and does not begin with "al", "as", "aS", or "ad". *Name* is case sensitive.

EquivalentLine

Specifies the alias equivalent. *EquivalentLine* is case sensitive. You must add at least one space between *Name* and *EquivalentLine*. The number of spaces between these two parameters is not important. The alias equivalent never contains leading spaces. After these spaces, *EquivalentLine* includes the rest of the line. Semicolons, quotation marks, and spaces are treated as literal characters, and trailing spaces are included.

EquivalentPhrase

Specifies the alias equivalent. *EquivalentPhrase* is case sensitive. You must add at least one space between *Name* and *EquivalentPhrase*. The number of spaces between these two parameters is not important. The alias equivalent never contains leading spaces.

You can enclose *EquivalentPhrase* in quotation marks (""). Regardless of whether you use quotation marks, *EquivalentPhrase* can contain spaces, commas, and single quotation marks (''). If you enclose *EquivalentPhrase* in quotation marks, it can include semicolons, but not additional quotation marks. If you do not enclose *EquivalentPhrase* in quotation marks, it can include quotation marks in any location other than the first character, but it

cannot include semicolons. Trailing spaces are included regardless of whether you use quotation marks.

/e

Sets the alias equivalent equal to the environment variable that *EnvironmentVariable* specifies.

EnvironmentVariable

Specifies the environment variable that is used to determine the alias equivalent. The debugger's environment is used, not the target's. If you started the debugger at a Command Prompt window, the environment variables in that window are used.

/ma

Sets the alias equivalent equal to the null-terminated ASCII string that begins at *Address*.

/mu

Sets the alias equivalent equal to the null-terminated Unicode string that begins at *Address*.

/msa

Sets the alias equivalent equal to the ANSI_STRING structure that is located at *Address*.

/msu

Sets the alias equivalent equal to the UNICODE_STRING structure that is located at *Address*.

Address

Specifies the location of the virtual memory that is used to determine the alias equivalent.

/x

Sets the alias equivalent equal to the 64-bit value of *Expression*.

Expression

Specifies the expression to evaluate. This value becomes the alias equivalent. For more information about the syntax, see [Numerical Expression Syntax](#).

/f

Sets the alias equivalent equal to the contents of the *File* file. You should always use the /f switch together with **aS**, not with **as**.

File

Specifies the file whose contents become the alias equivalent. *File* can contain spaces, but you should never enclose *File* in quotation marks. If you specify an invalid file, you receive an "Out of memory" error message.

/c

Sets the alias equivalent equal to the output of the commands that *CommandString* specifies. The alias equivalent includes carriage returns if they are present within the command display and a carriage return at the end of the display of each command (even if you specify only one command).

CommandString

Specifies the commands whose outputs become the alias equivalent. This string can include any number of commands that are separated by semicolons.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use aliases, see [Using Aliases](#).

Remarks

If you do not use any switches, the **aS** command uses the rest of the line as the alias equivalent.

You can end the **aS** command by a semicolon. This technique is useful in a script when you have to put all commands on a single line. Note that if the portion of the line after the semicolon requires expansion of the alias, you must enclose that second portion of the line in a new block. The following example produces the expected output, 0x6.

```
dbgcmd
```

```
0:001> aS /x myAlias 5 + 1; .block{.echo myAlias}  
0x6
```

If you omit the new block, you do not get the expected output. That is because the expansion of a newly set alias does not happen until a new code block is entered. In the

following example, the new block is omitted, and the output is the text "myAlias" instead of the expected value 0x6.

```
dbgcmd  
0:001> aS /x myAlias 5 + 1; .echo myAlias  
myAlias
```

For more information about using aliases in scripts, see [Using Aliases](#).

If you use a **/e**, **/ma**, **/mu**, **/msa**, **/msu**, or **/x** switch, the **as** and **aS** commands work the same and the command ends if a semicolon is encountered.

If *Name* is already the name of an existing alias, that alias is redefined.

You can use the **as** or **aS** command to create or change any user-named alias. But you cannot use the command to control a fixed-name alias (\$u0 to \$u9).

You can use the **/ma**, **/mu**, **/msa**, **/msu**, **/f**, and **/c** switches to create an alias that contains carriage returns. However, you cannot use an alias that contains carriage returns to execute multiple commands in sequence. Instead, you must use semicolons.

ba (Break on Access)

Article • 02/12/2024

The **ba** command sets a processor breakpoint (often called, less accurately, a *data breakpoint*). This breakpoint is triggered when the specified memory is accessed.

User-Mode

```
dbgcmd
```

```
[~Thread] ba[ID] Access Size [Options] [Address [Passes]] ["CommandString"]
```

Kernel-Mode

```
dbgcmd
```

```
ba[ID] Access Size [Options] [Address [Passes]] ["CommandString"]
```

Parameters

Thread

Specifies the thread that the breakpoint applies to. For more information about syntax, see [Thread Syntax](#). You can specify threads only in user mode.

ID

Specifies an optional number that identifies the breakpoint. If you do not specify *ID*, the first available breakpoint number is used. You cannot add space between **ba** and the ID number. Each processor supports only a limited number of processor breakpoints, but there is no restriction on the value of the *ID* number. If you enclose *ID* in square brackets ([]), *ID* can include any expression. For more information about the syntax, see [Numerical Expression Syntax](#).

Access

Specifies the type of access that satisfies the breakpoint. This parameter can be one of the following values.

[] [Expand table](#)

Option	Action
e (execute)	Breaks into the debugger when the CPU retrieves an instruction from the specified

Option	Action
	address.
r (read/write)	Breaks into the debugger when the CPU reads or writes at the specified address.
w (write)	Breaks into the debugger when the CPU writes at the specified address.
i (i/o)	(Kernel mode only, x86-based systems only) Breaks into the debugger when the I/O port at the specified <i>Address</i> is accessed.

Size

Specifies the size of the location, in bytes, to monitor for access. On an x86-based processor, this parameter can be 1, 2, or 4. However, if Access equals e, Size must be 1.

On an x64-based processor, this parameter can be 1, 2, 4, or 8. However, if Access equals e, Size must be 1.

Options Specifies breakpoint options. You can use any number of the following options, except as indicated:

/1

Creates a "one-shot" breakpoint. After this breakpoint is triggered, the breakpoint is permanently removed from the breakpoint list.

/p *EProcess*

(Kernel mode only) Specifies a process that is associated with this breakpoint. *EProcess* should be the actual address of the EPROCESS structure, not the PID. The breakpoint is triggered only if it is encountered in the context of this process.

/t *ETHread*

(Kernel mode only) Specifies a thread that is associated with this breakpoint. *ETHread* should be the actual address of the ETHREAD structure, not the thread ID. The breakpoint is triggered only if it is encountered in the context of this thread. If you use /p *EProcess* and /t *ETHread*, you can enter them in either order.

/c *MaxCallStackDepth*

Causes the breakpoint to be active only when the call stack depth is less than *MaxCallStackDepth*. You cannot combine this option together with /C.

/C *MinCallStackDepth*

Causes the breakpoint to be active only when the call stack depth is larger than *MinCallStackDepth*. You cannot combine this option together with /c.

/w dx object expression Sets a conditional breakpoint based on the boolean value returned by dx object expression. The argument is a data model (dx) expression which evaluates to true (matches condition – break) or false (does not match condition – do not break).

This example sets a conditional breakpoint based on the value of globalVariable. This allows an access breakpoint, for instance, to check the value that was written when determining if the debugger should break in.

```
dbgcmd  
ba w 4 /w "mymodule!globalVariable == 4" mymodule!globalVariable
```

This example shows how to set a breakpoint using JavaScript.

```
dbgcmd  
ba w 4 /w "@$scriptContents.myFunc(mymodule!globalVariable)"  
mymodule!globalVariable
```

For more information on debugger objects, see [dx \(Display Debugger Object Model Expression\)](#).

For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Address

Specifies any valid address. If the application accesses memory at this address, the debugger stops execution and displays the current values of all registers and flags. This address must be an offset and suitably aligned to match the *Size* parameter. (For example, if *Size* is 4, *Address* must be a multiple of 4.) If you omit *Address*, the current instruction pointer is used. For more information about the syntax, see [Address and Address Range Syntax](#).

Passes

Specifies the number of times the breakpoint is passed by until it activates. This number can be any 16-bit value. The number of times the program counter passes through this point without breaking is one less than the value of this number. Therefore, omitting this number is the same as setting it equal to 1. Note also that this number counts only the times that the application *executes* past this point. Stepping or tracing past this point does not count. After the full count is reached, you can reset this number only by clearing and resetting the breakpoint.

CommandString

Specifies a list of commands to execute every time that the breakpoint is encountered the specified number of times. These commands are executed only if the breakpoint is hit after you issue a [g \(Go\)](#) command, instead of after a [t \(Trace\)](#) or [p \(Step\)](#) command. Debugger commands in *CommandString* can include parameters.

You must enclose this command string in quotation marks, and you should separate multiple commands by semicolons. You can use standard C control characters (such as `\n` and `\\"`). Semicolons that are contained in second-level quotation marks (`\\"`) are interpreted as part of the embedded quoted string.

This parameter is optional.

Environment

[] [Expand table](#)

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information on processor breakpoints, see [Processor Breakpoints \(ba Breakpoints\)](#). For more information about and examples of using breakpoints, other breakpoint commands and methods of controlling breakpoints, and information about how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

The debugger uses the *ID* number to refer to the breakpoint in later [bc \(Breakpoint Clear\)](#), [bd \(Breakpoint Disable\)](#), and [be \(Breakpoint Enable\)](#) commands.

Use the [bl \(Breakpoint List\)](#) command to list all existing breakpoints, their ID numbers, and their status.

Use the [.bpcmds \(Display Breakpoint Commands\)](#) command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

Each processor breakpoint has a size associated with it. For example, a **w** (write) processor breakpoint could be set at the address 0x70001008 with a size of four bytes. This would monitor the block of addresses from 0x70001008 to 0x7000100B, inclusive. If this block of memory is written to, the breakpoint will be triggered.

It can happen that the processor performs an operation on a memory region that *overlaps* with, but is not identical to, the specified region. In this example, a single write operation that includes the range 0x70001000 to 0x7000100F, or a write operation that includes only the byte at 0x70001009, would be an overlapping operation. In such a situation, whether the breakpoint is triggered is processor-dependent. You should consult the processor manual for specific details. To take one specific instance, on an x86 processor, a read or write breakpoint is triggered whenever the accessed range overlaps the breakpoint range.

Similarly, if an **e** (execute) breakpoint is set on the address 0x00401003, and then a two-byte instruction spanning the addresses 0x00401002 and 0x00401003 is executed, the result is processor-dependent. Again, consult the processor architecture manual for details.

The processor distinguishes between breakpoints set by a user-mode debugger and breakpoints set by a kernel-mode debugger. A user-mode processor breakpoint does not affect any kernel-mode processes. A kernel-mode processor breakpoint might or might not affect a user-mode process, depending on whether the user-mode code is using the debug register state and whether there is a user-mode debugger that is attached.

To apply the current process' existing data breakpoints to a different register context, use the [.apply_dbp \(Apply Data Breakpoint to Context\)](#) command.

On a multiprocessor computer, each processor breakpoint applies to all processors. For example, if the current processor is 3 and you use the command `ba e1 MyAddress` to put a breakpoint at MyAddress, any processor -- not only processor 3 -- that executes at that address triggers the breakpoint. (This holds for software breakpoints as well.)

You cannot create multiple processor breakpoints at the same address that differ only in their *CommandString* values. However, you can create multiple breakpoints at the same address that have different restrictions (for example, different values of the **/p**, **/t**, **/c**, and **/C** options).

For more details on processor breakpoints, and additional restrictions that apply to them, see [Processor Breakpoints \(ba Breakpoints\)](#).

The following examples show the **ba** command. The following command sets a breakpoint for read access on 4 bytes of the variable myVar.

```
dbgcmd  
0:000> ba r4 myVar
```

The following command adds a breakpoint on all serial ports with addresses from 0x3F8 through 0x3FB. This breakpoint is triggered if anything is read or written to these ports.

```
dbgcmd  
kd> ba i4 3f8
```

bc (Breakpoint Clear)

Article • 10/25/2023

The **bc** command permanently removes previously set breakpoints from the system.

```
dbgcmd
```

```
bc Breakpoints
```

Parameters

Breakpoints

Specifies the ID numbers of the breakpoints to remove. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a [numeric expression](#) for an ID, enclose it in brackets ([]). If you want to use a [string with wildcard characters](#) to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

Use the [bl \(Breakpoint List\)](#) command to list all existing breakpoints, their ID numbers, and their status.

Use the [**.bpcmds** \(Display Breakpoint Commands\)](#) command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

bd (Breakpoint Disable)

Article • 10/25/2023

The **bd** command disables, but does not delete, previously set breakpoints.

dbgcmd

bd Breakpoints

Parameters

Breakpoints

Specifies the ID numbers of the breakpoints to disable. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a [numeric expression](#) for an ID, enclose it in brackets ([]). If you want to use a [string with wildcard characters](#) to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

When a breakpoint is disabled, the system does not check whether the conditions that are specified in the breakpoint are valid.

Use the [**be** \(Breakpoint Enable\)](#) command to re-enable a disabled breakpoint.

Use the [**bl** \(Breakpoint List\)](#) command to list all existing breakpoints, their ID numbers, and their status.

Use the [**.bpcmds** \(Display Breakpoint Commands\)](#) command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

be (Breakpoint Enable)

Article • 10/25/2023

The **be** command restores one or more breakpoints that were previously disabled.

```
dbgcmd
```

```
  be Breakpoints
```

Parameters

Breakpoints

Specifies the ID numbers of the breakpoints to enable. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a [numeric expression](#) for an ID, enclose it in brackets ([]). If you want to use a [string with wildcard characters](#) to match a breakpoint's symbolic name, enclose it in quotation marks (" ").

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

Use the [bl \(Breakpoint List\)](#) command to list all existing breakpoints, their ID numbers, and their status.

Use the [**.bpcmds** \(Display Breakpoint Commands\)](#) command to list all existing breakpoints, their ID numbers, and the commands that were used to create them.

bl (Breakpoint List)

Article • 10/25/2023

The **bl** command lists information about existing breakpoints.

dbgcmd

bl [/L] [Breakpoints]

Parameters

/L

Forces **bl** to always display breakpoint addresses instead of showing source file and line numbers.

Breakpoints

Specifies the ID numbers of the breakpoints to list. If you omit *Breakpoints*, the debugger lists all breakpoints. You can specify any number of breakpoints. You must separate multiple IDs by spaces or commas. You can specify a range of breakpoint IDs by using a hyphen (-). You can use an asterisk (*) to indicate all breakpoints. If you want to use a [numeric expression](#) for an ID, enclose it in brackets ([]). If you want to use a [string with wildcard characters](#) to match a breakpoint's symbolic name, enclose it in quotation marks ("").

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

For each breakpoint, the command displays the following information:

- The breakpoint ID. This ID is a decimal number that you can use to refer to the breakpoint in later commands.
- The breakpoint status. The status can be **e** (enabled) or **d** (disabled).
- (Unresolved breakpoints only) The letter "u" appears if the breakpoint is unresolved. That is, the breakpoint does not match a symbolic reference in any currently loaded module. For information about these breakpoints, see [Unresolved Breakpoints \(bu Breakpoints\)](#).
- The virtual address or symbolic expression that makes up the breakpoint location. If you enabled source line number loading, the **bl** command displays file and line number information instead of address offsets. If the breakpoint is unresolved, the address is omitted here and appears at the end of the listing instead.
- (Data breakpoints only) Type and size information are displayed for data breakpoints. The types can be **e** (execute), **r** (read/write), **w** (write), or **i** (input/output). These types are followed with the size of the block, in bytes. For information about these breakpoints, see [Processor Breakpoints \(ba Breakpoints\)](#).
- The number of passes that remain until the breakpoint is activated, followed by the initial number of passes in parentheses. For more information about this kind of breakpoint, see the description of the *Passes* parameter in [bp, bu, bm \(Set Breakpoint\)](#).
- The associated process and thread. If thread is given as three asterisks (***)¹, this breakpoint is not a thread-specific breakpoint.
- The module and function, with offset, that correspond to the breakpoint address. If the breakpoint is unresolved, the breakpoint address appears here instead, in parentheses. If the breakpoint is set on a valid address but symbol information is missing, this field is blank.
- The command that is automatically executed when this breakpoint is hit. This command is displayed in quotation marks.

If you are not sure what command was used to set an existing breakpoint, use [.bpcmds \(Display Breakpoint Commands\)](#) to list all breakpoints along with the commands that were used to create them.

The following example shows the output of a **bl** command.

Example

```
dbgcmd  
0:000> b1  
0 e 010049e0      0001 (0001)  0:**** stst!main
```

This output contains the following information:

- The breakpoint ID is **0**.
- The breakpoint status is **e** (enabled).
- The breakpoint is not unresolved (there is no **u** in the output).
- The virtual address of the breakpoint is **010049e0**.
- The breakpoint is active on the first pass through the code and the code has not yet been executed under the debugger. This information is indicated by a value of **1 (0001)** in the "passes remaining" counter and a value of **1 ((0001))** in the initial passes counter.
- This breakpoint is not a thread-specific breakpoint (***)�.
- The breakpoint is set on **main** in the **stst** module.

bp, bu, bm (Set Breakpoint)

Article • 10/25/2023

The **bp**, **bu**, and **bm** commands set one or more software breakpoints. You can combine locations, conditions, and options to set different kinds of software breakpoints.

User-Mode

dbgcmd

```
[~Thread] bp[ID] [Options] [Address [Passes]] ["CommandString"]
[~Thread] bu[ID] [Options] [Address [Passes]] ["CommandString"]
[~Thread] bm [Options] SymbolPattern [Passes] ["CommandString"]
```

Kernel-Mode

dbgcmd

```
bp[ID] [Options] [Address [Passes]] ["CommandString"]
bu[ID] [Options] [Address [Passes]] ["CommandString"]
bm [Options] SymbolPattern [Passes] ["CommandString"]
```

Parameters

Thread

Specifies the thread that the breakpoint applies to. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode. If you do not specify a thread, the breakpoint applies to all threads.

ID

Specifies a decimal number that identifies a breakpoint.

The debugger assigns the *ID* when it creates the breakpoint, but you can change it by using the [br \(Breakpoint Rerumber\)](#) command. You can use the *ID* to refer to the breakpoint in later debugger commands. To display the *ID* of a breakpoint, use the [bl \(Breakpoint List\)](#) command.

When you use *ID* in a command, do not type a space between the command (**bp** or **bu**) and the ID number.

The *ID* parameter is always optional. If you do not specify *ID*, the debugger uses the first available breakpoint number. In kernel mode, you can set only 32 breakpoints. In user

mode, you can set any number of breakpoints. In either case, there is no restriction on the value of the *ID* number. If you enclose *ID* in square brackets ([]), *ID* can include any expression. For more information about the syntax, see [Numerical Expression Syntax](#).

Options Specifies breakpoint options. You can specify any number of the following options, except as indicated:

/1

Creates a "one-shot" breakpoint. After this breakpoint is triggered, it is deleted from the breakpoint list.

/p EProcess

(Kernel-mode only) Specifies a process that is associated with this breakpoint. *EProcess* should be the actual address of the EPROCESS structure, not the PID. The breakpoint is triggered only if it is encountered in the context of this process.

/t ETthread

(Kernel-mode only) Specifies a thread that is associated with this breakpoint. *ETthread* should be the actual address of the ETHREAD structure, not the thread ID. The breakpoint is triggered only if it is encountered in the context of this thread. If you use **/p EProcess** and **/t ETthread**, you can enter them in any order.

/c MaxCallStackDepth

Activates the breakpoint only when the call stack depth is less than *MaxCallStackDepth*. You cannot use this option together with **/C**.

/C MinCallStackDepth

Activates the breakpoint only when the call stack depth is larger than *MinCallStackDepth*. You cannot use this option together with **/c**.

/a

(For **bm** only) Sets breakpoints on all of the specified locations, whether they are in data space or code space. Because breakpoints on data can cause program failures, use this option only on locations that are known to be safe.

/d

(For **bm** only) Converts the breakpoint locations to addresses. Therefore, if the code is moved, the breakpoints remain at the same address, instead of being set according to *SymbolPattern*. Use **/d** to avoid reevaluating changes to breakpoints when modules are loaded or unloaded.

/

(For **bm** only) Includes parameter list information in the symbol string that *SymbolString* defines.

This feature enables you to set breakpoints on overloaded functions that have the same name but different parameter lists. For example, `bm / myFunc` sets breakpoints on both `myFunc(int a)` and `myFunc(char a)`. Without `"/"`, a breakpoint that is set on `myFunc` fails because it does not indicate which `myFunc` function the breakpoint is intended for.

/w dx object expression Sets a conditional breakpoint based on the boolean value returned by `dx` object expression. The argument is a data model (`dx`) expression which evaluates to true (matches condition – break) or false (does not match condition – do not break).

This example sets a conditional breakpoint based on the value of `localVariable`.

```
dbgcmd  
bp /w "localVariable == 4" mymodule!myfunction
```

This example shows how to set a breakpoint using JavaScript.

```
dbgcmd  
bp /w "@$scriptContents.myFunc(localVariable)" @rip
```

For more information on debugger objects, see [dx \(Display Debugger Object Model Expression\)](#).

For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Address

Specifies the first byte of the instruction where the breakpoint is set. If you omit *Address*, the current instruction pointer is used. For more information about the syntax, see [Address and Address Range Syntax](#).

Passes

Specifies the number of the execution pass that the breakpoint is activated on. The debugger skips the breakpoint location until it reaches the specified pass. The value of *Passes* can be any 16-bit or 32-bit value.

By default, the breakpoint is active the first time that the application executes the code that contains the breakpoint location. This default situation is equivalent to a value of **1** for *Passes*. To activate the breakpoint only after the application executes the code at least one time, enter a value of **2** or more. For example, a value of **2** activates the breakpoint the second time that the code is executed.

This parameter creates a counter that is decremented on each pass through the code. To see the initial and current values of the *Passes* counter, use [bl \(Breakpoint List\)](#).

The *Passes* counter is decremented only when the application *executes* past the breakpoint in response to a [g \(Go\)](#) command. The counter is not decremented if you are stepping through the code or tracing past it. When the *Passes* counter reaches 1, you can reset it only by clearing and resetting the breakpoint.

CommandString

Specifies a list of commands that are executed every time that the breakpoint is encountered the specified number of times. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as `\n` and `\\"`). Semicolons that are contained in second-level quotation marks (`\\"`) are interpreted as part of the embedded quoted string.

The *CommandString* commands are executed only if the breakpoint is reached while the application is *executing* in response to a [g \(Go\)](#) command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as `g` or `t`) ends the execution of the command list.

SymbolPattern

Specifies a pattern. The debugger tries to match this pattern to existing symbols and to set breakpoints on all pattern matches. *SymbolPattern* can contain a variety of wildcard characters and specifiers. For more information about this syntax, see [String Wildcard Syntax](#). Because these characters are being matched to symbols, the match is not case sensitive, and a single leading underscore (`_`) represents any quantity of leading underscores.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

The **bp**, **bu**, and **bm** commands set new breakpoints, but they have different characteristics:

- The **bp** (**Set Breakpoint**) command sets a new breakpoint at the *address* of the breakpoint location that is specified in the command. If the debugger cannot resolve the address expression of the breakpoint location when the breakpoint is set, the **bp** breakpoint is automatically converted to a **bu** breakpoint. Use a **bp** command to create a breakpoint that is no longer active if the module is unloaded.
- The **bu** (**Set Unresolved Breakpoint**) command sets a *deferred* or *unresolved* breakpoint. A **bu** breakpoint is set on a symbolic reference to the breakpoint location that is specified in the command (not on an address) and is activated whenever the module with the reference is resolved. For more information about these breakpoints, see [Unresolved Breakpoints \(bu Breakpoints\)](#).
- The **bm** (**Set Symbol Breakpoint**) command sets a new breakpoint on symbols that match a specified pattern. This command can create more than one breakpoint. By default, after the pattern is matched, **bm** breakpoints are the same as **bu** breakpoints. That is, **bm** breakpoints are deferred breakpoints that are set on a symbolic reference. However, a **bm /d** command creates one or more **bp** breakpoints. Each breakpoint is set on the address of a matched location and does not track module state.

If you are not sure what command was used to set an existing breakpoint, use [.bpcmds \(Display Breakpoint Commands\)](#) to list all breakpoints along with the commands that were used to create them.

There are three primary differences between **bp** breakpoints and **bu** breakpoints:

- A **bp** breakpoint location is always converted to an address. If a module change moves the code at which a **bp** breakpoint was set, the breakpoint remains at the same address. On the other hand, a **bu** breakpoint remains associated with the symbolic value (typically a symbol plus an offset) that was used, and it tracks this symbolic location even if its address changes.

- If a **bp** breakpoint address is found in a loaded module, and if that module is later unloaded, the breakpoint is removed from the breakpoint list. On the other hand, **bu** breakpoints persist after repeated unloads and loads.
- Breakpoints that you set with **bp** are not saved in WinDbg [workspaces](#). Breakpoints that are set with **bu** are saved in workspaces.

The **bm** command is useful when you want to use wildcard characters in the symbol pattern for a breakpoint. The **bm SymbolPattern** syntax is equivalent to using [x SymbolPattern](#) and then using **bu** on each result. For example, to set breakpoints on all of the symbols in the *Myprogram* module that begin with the string "mem," use the following command.

Example

```
dbgcmd

0:000> bm myprogram!mem*
4: 0040d070 MyProgram!memcpy
5: 0040c560 MyProgram!memmove
6: 00408960 MyProgram!memset
```

Because the **bm** command sets software breakpoints (not processor breakpoints), it automatically excludes data location when it sets breakpoints to avoid corrupting the data.

It is possible to specify a data address rather than a program address when using the **bp** or **bm /a** commands. However, even if a data location is specified, these commands create software breakpoints, not processor breakpoints. If a software breakpoint is placed in program data instead of executable code, it can lead to data corruption. Therefore you should use these commands in a data location only if you are certain that the memory stored in that location will be used as executable code and not as program data. Otherwise, you should use the [ba \(Break on Access\)](#) command instead. For more details, see [Processor Breakpoints \(ba Breakpoints\)](#).

For details on how to set a breakpoint on a location specified by a more complicated syntax, such as a member of a C++ public class, or an arbitrary text string containing otherwise restricted characters, see [Breakpoint Syntax](#).

If a single logical source line spans multiple physical lines, the breakpoint is set on the last physical line of the statement or call. If the debugger cannot set a breakpoint at the requested position, it puts the breakpoint in the next allowed position.

If you specify *Thread*, breakpoints are set on the specified threads. For example, the `~*bp` command sets breakpoints on all threads, `~#bp` sets a breakpoint on the thread that causes the current exception, and `~123bp` sets a breakpoint on thread 123. The `~bp` and `~.bp` commands both set a breakpoint on the current thread.

When you are debugging a multiprocessor system in kernel mode, breakpoints that you set by using `bp` or [ba \(Break on Access\)](#) apply to all processors. For example, if the current processor is 3 and you type `bp MemoryAddress` to put a breakpoint at `MemoryAddress`. Any processor that is executing at that address (not only processor 3) causes a breakpoint trap.

The `bp`, `bu`, and `bm` commands set software breakpoints by replacing the processor instruction with a break instruction. To debug read-only code or code that cannot be changed, use a `ba e` command, where `e` represents execute-only access.

The following command sets a breakpoint 12 bytes past the beginning of the function `MyTest`. This breakpoint is ignored for the first six passes through the code, but execution stops on the seventh pass through the code.

```
dbgcmd  
0:000> bp MyTest+0xb 7
```

The following command sets a breakpoint at `RtlRaiseException`, displays the `eax` register, displays the value of the symbol `MyVar`, and continues.

```
dbgcmd  
kd> bp ntdll!RtlRaiseException "r eax; dt MyVar; g"
```

The following two `bm` commands set three breakpoints. When the commands are executed, the displayed result does not distinguish between breakpoints created with the `/d` switch and those created without it. The [.bpcmds \(Display Breakpoint Commands\)](#) can be used to distinguish between these two types. If the breakpoint was created by `bm` without the `/d` switch, the `.bpcmds` display indicates the breakpoint type as `bu`, followed by the evaluated symbol enclosed in the `@!""` token (which indicates it is a literal symbol and not a numeric expression or register). If the breakpoint was created by `bm` with the `/d` switch, the `.bpcmds` display indicates the breakpoint type as `bp`.

```
dbgcmd
```

```
0:000> bm myprog!openf*
0: 00421200 @!"myprog!openFile"
1: 00427800 @!"myprog!openFilter"

0:000> bm /d myprog!closef*
2: 00421600 @!"myprog!closeFile"

0:000> .bpcmds
bu0 @!"myprog!openFile";
bu1 @!"myprog!openFilter";
bp2 0x00421600 ;
```

br (Breakpoint Rerumber)

Article • 10/25/2023

The **br** command renames one or more breakpoints.

dbgcmd

```
br OldID NewID [OldID2 NewID2 ...]
```

Parameters

OldID

Specifies the current ID number of the breakpoint.

NewID

Specifies a new number that becomes the ID of the breakpoint.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

You can use the **br** command to rename any number of breakpoints at the same time. For each breakpoint, list the old ID and the new ID, in that order, as parameters to **br**.

If there is already a breakpoint with an ID equal to *NewID*, the command fails and an error message is displayed.

bs (Update Breakpoint Command)

Article • 10/25/2023

The **bs** command changes the command executed when the specified breakpoint is encountered.

dbgcmd

```
bs ID ["CommandString"]
```

Parameters

ID

Specifies the ID number of the breakpoint.

CommandString

Specifies the new list of commands to be executed every time that the breakpoint is encountered. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as `\n` and `\\"`). Semicolons that are contained in second-level quotation marks (`\"`) are interpreted as part of the embedded quoted string.

The *CommandString* commands are executed only if the breakpoint is reached while the application is executing in response to a **g (Go)** command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as **g** or **t**) ends the execution of the command list.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

If the *CommandString* is not specified, any commands already set on the breakpoint are removed.

bsc (Update Conditional Breakpoint)

Article • 10/25/2023

The **bsc** command changes the condition under which a breakpoint occurs or changes the command executed when the specified conditional breakpoint is encountered.

dbgcmd

```
bsc ID Condition ["CommandString"]
```

Parameters

ID

Specifies the ID number of the breakpoint.

Condition

Specifies the condition under which the breakpoint should be triggered.

CommandString

Specifies the new list of commands to be executed every time that the breakpoint is encountered. You must enclose the *CommandString* parameter in quotation marks. Use semicolons to separate multiple commands.

Debugger commands in *CommandString* can include parameters. You can use standard C-control characters (such as `\n` and `\"`). Semicolons that are contained in second-level quotation marks (`\"`) are interpreted as part of the embedded quoted string.

The *CommandString* commands are executed only if the breakpoint is reached while the application is executing in response to a **g (Go)** command. The commands are not executed if you are stepping through the code or tracing past this point.

Any command that resumes program execution after a breakpoint (such as **g** or **t**) ends the execution of the command list.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, and how to set breakpoints in user space from a kernel debugger, see [Using Breakpoints](#). For more information about conditional breakpoints, see [Setting a Conditional Breakpoint](#).

Remarks

If the *CommandString* is not specified, any commands already set on the breakpoint are removed.

The same effect can be achieved by using the [bs \(Update Breakpoint Command\)](#) command with the following syntax:

```
dbgcmd  
bs ID "j Condition 'CommandString'; 'gc'"
```

c (Compare Memory)

Article • 10/25/2023

The **c** command compares the values held in two memory areas.

dbgcmd

c Range Address

Parameters

Range

The first of the two memory ranges to be compared. For more syntax details, see [Address and Address Range Syntax](#).

Address

The starting address of the second memory range to be compared. The size of this range will be the same as that specified for the first range. For more syntax details, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

If the two areas are not identical, the debugger will display all memory addresses in the first range where they do not agree.

As an example, consider the following code:

C++

```
void main()
{
    char rgBuf1[100];
    char rgBuf2[100];

    memset(rgBuf1, 0xCC, sizeof(rgBuf1));
    memset(rgBuf2, 0xCC, sizeof(rgBuf2));

    rgBuf1[42] = 0xFF;
}
```

To compare **rgBuf1** and **rgBuf2**, use either of the following commands:

dbgcmd

```
0:000> c rgBuf1 (rgBuf1+0n100) rgBuf2
0:000> c rgBuf1 L 0n100 rgBuf2
```

d, da, db, dc, dd, dD, df, dp, dq, du, dw (Display Memory)

Article • 10/25/2023

The **d*** commands display the contents of memory in the given range.

```
dbgcmd

d{a|b|c|d|D|f|p|q|u|w|W} [Options] [Range]
dy{b|d} [Options] [Range]
d [Options] [Range]
```

Parameters

Options

Specifies one or more display options. You can include any of the following options, but no more than one **/p*** option.

/cWidth

Specifies the number of columns to use in the display. If you don't specify this option, the default number of columns depends on the display type.

/p

(Kernel mode only) Uses physical memory addresses for the display. The range specified by **Range** is taken from physical memory rather than virtual memory.

/p[c]

(Kernel mode only) Same as **/p**, except that cached memory is read. Include the brackets around **c**.

/p[uc]

(Kernel mode only) Same as **/p**, except that uncached memory is read. Include the brackets around **uc**.

/p[wc]

(Kernel mode only) Same as **/p**, except that write-combined memory is read. Include the brackets around **wc**.

Range

Specifies the memory area to display. For more syntax details, see [Address and address range syntax](#). If you omit **Range**, the command displays memory starting at the ending location of the last display command. If you omit **Range** and there's no previous display command, the display begins at the current instruction pointer.

Environment

Modes: user mode, kernel mode

Targets: live, crash dump

Platforms: all

Additional information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and writing memory](#).

Remarks

Each line displayed includes the address of the first byte in the line followed by the contents of memory at that and following locations.

If you omit **Range**, the command displays memory starting at the ending location of the last display command. This approach allows you to continuously scan through memory.

This command exists in the following forms. The second characters of the **dd**, **dD**, , and **dW** commands are case-sensitive, as are the third characters of the **dyb** and **dyd** commands.

Command	Display
d	This command displays data in the same format as the most recent d* command. If no previous d* command has been issued, d* has the same effect as db . Notice that d repeats the most recent command that began with d . These commands include dda , ddp , ddu , dpa , dpp , dpu , dqa , dqp , dqu , dds , dps , dqs , ds , dS , dg , dl , dt , dv , and the display commands in this article. If the parameters given after d aren't appropriate, errors might result.
da	ASCII characters. Each line displays up to 48 characters. The display continues until the first null byte or until all characters in range have been displayed. All

Command	Display
	nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.).
db	Byte values and ASCII characters. Each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The eighth and ninth hexadecimal values are separated by a hyphen (-). All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.). The default count is 128 bytes.
dc	Double-word values (4 bytes) and ASCII characters. Each display line shows the address of the first word in the line and up to eight hexadecimal word values and their ASCII equivalent. The default count is 32 DWORDs (128 bytes).
dd	Double-word values (4 bytes). The default count is 32 DWORDs (128 bytes).
dD	Double-precision floating-point numbers (8 bytes). The default count is 15 numbers (120 bytes).
df	Single-precision floating-point numbers (4 bytes). The default count is 16 numbers (64 bytes).
dp	Pointer-sized values. This command is equivalent to dd or dq , depending on whether the target computer processor architecture is 32-bit or 64-bit, respectively. The default count is 32 DWORDs or 16 quad-words (128 bytes).
dq	Quad-word values (8 bytes). The default count is 16 quad-words (128 bytes).
du	Unicode characters. Each line displays up to 48 characters. The display continues until the first null byte or until all characters in range have been displayed. All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.).
dw	Word values (2 bytes). Each display line shows the address of the first word in the line and up to eight hexadecimal word values. The default count is 64 words (128 bytes).
dW	Word values (2 bytes) and ASCII characters. Each display line shows the address of the first word in the line and up to eight hexadecimal word values. The default count is 64 words (128 bytes).
dyb	Binary values and byte values. The default count is 32 bytes.
dyd	Binary values and double-word values (4 bytes). The default count is 8 DWORDs (32 bytes).

If you attempt to display an invalid address, its contents are shown as question marks (?).

dda, ddp, ddu, dpa, dpp, dpu, dqa, dqp, dqu (Display Referenced Memory)

Article • 10/25/2023

The **dda**, **ddp**, **ddu**, **dpa**, **dpp**, **dpu**, **dqa**, **dqp**, and **dqu** commands display the pointer at the specified location, dereference that pointer, and then display the memory at the resulting location in a variety of formats.

dbgcmd

```
ddp [Options] [Range]
dqp [Options] [Range]
dpp [Options] [Range]
dda [Options] [Range]
dqa [Options] [Range]
dpa [Options] [Range]
ddu [Options] [Range]
dqu [Options] [Range]
dpu [Options] [Range]
```

Parameters

Options Specifies one or more display options. Any of the following options can be included, except that no more than one **/p*** option can be indicated:

/cWidth

Specifies the number of columns to use in the display. If this is omitted, the default number of columns depends on the display type. Because of the way pointers are displayed by these commands, it is usually best to use the default of only one data column.

/p

(Kernel-mode only) Uses physical memory addresses for the display. The range specified by *Range* will be taken from physical memory rather than virtual memory.

/p[c]

(Kernel-mode only) Same as **/p**, except that cached memory will be read. The brackets around **c** must be included.

/p[uc]

(Kernel-mode only) Same as **/p**, except that uncached memory will be read. The brackets around **uc** must be included.

/p[wc]

(Kernel-mode only) Same as **/p**, except that write-combined memory will be read. The brackets around **wc** must be included.

Range

Specifies the memory area to display. For more syntax details, see [Address and Address Range Syntax](#). If you omit *Range*, the command will display memory starting at the ending location of the last display command. If *Range* is omitted and no previous display command has been used, the display begins at the current instruction pointer. If a simple address is given, the default range length is 128 bytes.

Environment

Modes: user mode, kernel mode

Targets: live, crash dump

Platforms: all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The second and third characters of this command are case-sensitive.

The second character of this command determines the pointer size used:

Command	Display
dd	32-bit pointers used
dq	64-bit pointers used
dp*	Standard pointer sizes used: 32-bit or 64-bit, depending on the target's processor architecture

The third character of this command determines how the dereferenced memory is displayed:

Command	Display
dp	Displays the contents of the memory referenced by the pointer in DWORD or QWORD format, depending on the pointer size of the target's processor architecture. If this value matches any known symbol, this symbol is displayed as well.
da	Displays the contents of the memory referenced by the pointer in ASCII character format.
d*u	Displays the contents of the memory referenced by the pointer in Unicode character format.

If line number information has been enabled, source file names and line numbers will be displayed when available.

dds, dps, dqs (Display Words and Symbols)

Article • 10/25/2023

The **dds**, **dps**, and **dqs** commands display the contents of memory in the given range. This memory is assumed to be a series of addresses in the symbol table. The corresponding symbols are displayed as well.

dbgcmd

```
dds [Options] [Range]
dqs [Options] [Range]
dps [Options] [Range]
```

Parameters

Options Specifies one or more display options. Any of the following options can be included, except that no more than one **/p*** option can be indicated:

/c Width

Specifies the number of columns to use in the display. If this is omitted, the default number of columns depends on the display type. Because of the way symbols are displayed by these commands, it is usually best to use the default of only one data column.

/p

(Kernel-mode only) Uses physical memory addresses for the display. The range specified by *Range* will be taken from physical memory rather than virtual memory.

/p[c]

(Kernel-mode only) Same as **/p**, except that cached memory will be read. The brackets around **c** must be included.

/p[uc]

(Kernel-mode only) Same as **/p**, except that uncached memory will be read. The brackets around **uc** must be included.

/p[wc]

(Kernel-mode only) Same as **/p**, except that write-combined memory will be read. The brackets around **wc** must be included.

Range

Specifies the memory area to display. For more syntax details, see [Address and Address Range Syntax](#). If you omit *Range*, the command will display memory starting at the ending location of the last display command. If *Range* is omitted and no previous display command has been used, the display begins at the current instruction pointer. If a simple address is given, the default range length is 128 bytes.

Environment

Modes: user mode, kernel mode

Targets: live, crash dump

Platforms: all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The second character of **dds** is case-sensitive. The third character of all these commands is case-sensitive.

The **dds** command displays double-word (4 byte) values like the **dd** command. The **dqs** command displays quad-word (8 byte) values like the **dq** command. The **dps** command displays pointer-sized values (4 byte or 8 byte, depending on the target computer's architecture) like the **dp** command.

Each of these words is treated as an address in the symbol table. The corresponding symbol information is displayed for each word.

If line number information has been enabled, source file names and line numbers will be displayed when available.

dg (Display Selector)

Article • 10/25/2023

The **dg** command shows the segment descriptor for the specified selector.

dbgcmd

```
dg FirstSelector [LastSelector]
```

Parameters

FirstSelector

Specifies the hexadecimal selector value of the first selector to be displayed.

LastSelector

Specifies the hexadecimal selector value of the last selector to be displayed. If this is omitted, only one selector will be displayed.

Environment

Modes	user mode, kernel mode
Targets	live, crash dump
Platforms	x86

Remarks

No more than 256 selectors can be displayed by this command.

Common selector values are:

Id	decimal	hex
KGDT_NULL	0	0x00
KGDT_R0_CODE	8	0x08
KGDT_R0_DATA	16	0x10
KGDT_R3_CODE	24	0x18

Id	decimal	hex
KGDT_R3_DATA	32	0x20
KGDT_TSS	40	0x28
KGDT_R0_PCR	48	0x30
KGDT_R3_TEB	56	0x38
KGDT_VDM_TILE	64	0x40
KGDT_LDT	72	0x48
KGDT_DF_TSS	80	0x50
KGDT_NMI_TSS	88	0x58

dl (Display Linked List)

Article • 10/25/2023

The **dl** command displays a LIST_ENTRY or SINGLE_LIST_ENTRY linked list.

dbgcmd

dl[b] Address MaxCount Size

Parameters

b

If this is included, the list is dumped in reverse order. (In other words, the debugger follows the **Blinks** instead of the **Flinks**.) This cannot be used with a SINGLE_LIST_ENTRY.

Address

The starting address of the list. For more syntax details, see [Address and Address Range Syntax](#).

MaxCount

Maximum number of elements to dump.

Size

Size of each element. This is the number of consecutive ULONG_PTRs that will be displayed for each element in the list.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

This list must be a LIST_ENTRY or SINGLE_LIST_ENTRY structure. If this is embedded in a larger structure, be sure that *Address* points to the linked list structure and not to the beginning of the outer structure.

The display begins with *Address*. Therefore, if you are supplying the address of a pointer that points to the beginning of the list, you should disregard the first element printed.

The *Address*, *MaxCount*, and *Size* parameters are in the current default radix. You can use the [n \(Set Number Base\)](#) command or the `0x` prefix to change the radix.

If the list loops back on itself, the dump will stop. If a null pointer is encountered, the dump will stop.

If you want to execute some command for each element of the list, use the [!list](#) extension.

ds, dS (Display String)

Article • 10/25/2023

The **ds** and **dS** commands display a STRING, ANSI_STRING, or UNICODE_STRING structures.

These commands do not display null-delimited character strings, but rather string structures.

If you have the address of the characters of a Unicode string then use the **du** command instead. Use the **da** command to display ASCII characters. For more information, see [d, da, db, dc, dd, dD, df, dp, dq, du, dw \(Display Memory\)](#).

```
dbgcmd
```

```
  d{s|S} [/c Width] [Address]
```

Parameters

s

Specifies that a STRING or ANSI_STRING structure is to be displayed. (This **s** is case-sensitive.)

S

Specifies that a UNICODE_STRING structure is to be displayed. (This **S** is case-sensitive.)

/c Width

Specifies the number of characters to display on each line. This number includes null characters, which will not be visible.

Address

The memory address where the where the UNICODE_STRING structure is stored.

For more syntax details, see [Address and Address Range Syntax](#). If omitted, the last address used in a display command is assumed.

Environment

Item	Description
Modes	User mode, kernel mode

Item	Description
Targets	Live, crash dump
Platforms	All

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

If you want to display Unicode strings in the Locals window or Watch window of WinDbg, you need to use the [.enable_unicode \(Enable Unicode Display\)](#) command first.

dt (Display Type)

Article • 10/25/2023

The **dt** command displays information about a local variable, global variable or data type. This can display information about simple data types, as well as structures and unions.

User-Mode Syntax

```
dbgcmd

dt [-DisplayOpts] [-SearchOpts] [module!]Name [[-SearchOpts] Field]
[Address] [-l List]
dt [-DisplayOpts] Address [-l List]
dt -h
```

Kernel-Mode Syntax

```
dbgcmd

[Processor] dt [-DisplayOpts] [-SearchOpts] [module!]Name [[-SearchOpts]
Field] [Address] [-l List]
dt [-DisplayOpts] Address [-l List]
dt -h
```

Parameters

Processor

Specifies the processor that is running the process containing the information needed. For more information, see [Multiprocessor Syntax](#). Processors can only be specified in kernel mode.

DisplayOpts

Specifies one or more of the options given in the following table. These options are preceded by a hyphen.

Option	Description
<code>-a[quantity]</code>	Show each array element on a new line, with its index. A total of <i>quantity</i> elements will be displayed. There must be no space between the <code>a</code> and the <i>quantity</i> . If <code>-a</code> is not followed by a digit, all items in the array are shown. The <code>-</code>

Option	Description
	a[<i>quantity</i>] switch should appear immediately before each type name or field name that you want displayed in this manner.
-b	Display blocks recursively. If a displayed structure contains substructures, it is expanded recursively to arbitrary depths and displayed in full. Pointers are expanded only if they are in the original structure, not in substructures.
-c	Compact output. All fields are displayed on one line, if possible. (When used with the -a switch, each array element takes one line rather than being formatted as a several-line block.)
-d	When used with a <i>Name</i> that is ended with an asterisk, display verbose output for all types that begin with <i>Name</i> . If <i>Name</i> does not end with an asterisk, display verbose output.
-e	Forces dt to enumerate types. This option is only needed if dt is mistakenly interpreting the <i>Name</i> value as an instance rather than as a type.
-i	Do not indent the subtypes.
-o	Omit offset values of the structure fields.
-p	<i>Address</i> is a physical address, rather than a virtual address.
-r[<i>depth</i>]	Recursively dumps the subtype fields. If <i>depth</i> is given, this recursion will stop after <i>depth</i> levels. The <i>depth</i> must be a digit between 1 and 9, and there must be no space between the r and the <i>depth</i> . The -r[<i>depth</i>] switch should appear immediately before the address.
-s <i>size</i>	Enumerate only those types whose size in bytes equals the value of <i>size</i> . The -s option is only useful when types are being enumerated. When -s is specified, -e is always implied as well.
-t	Enumerate types only.
-v	Verbose output. This gives additional information such as the total size of a structure and the number of its elements. When this is used along with the -y search option, all

Option	Description
	symbols are displayed, even those with no associated type information.

SearchOpts

Specifies one or more of the options given in the following table. These options are preceded by a hyphen.

Option	Description
-n	This indicates that the next parameter is a name. This should be used if the next item consists entirely of hexadecimal characters, because it will otherwise be taken as an address.
-y	This indicates that the next parameter is the beginning of the name, not necessarily the entire name. When -y is included, all matches are listed, followed by detailed information on the first match in the list. If -y is not included, only exact matches will be displayed.

module

An optional parameter specifying the module that defines this structure. If there is a local variable or type with the same name as a global variable or type, you should include *module* to specify that you mean the global variable. Otherwise, the **dt** command will display the local variable, even if the local variable is a case-insensitive match and the global variable is a case-sensitive match.

Name

Specifies the name of a type or global variable. If *Name* ends with an asterisk (*), a list of all matches is displayed. Thus, **dt A*** will list all data types, globals, and statics beginning with "A", but will not display the actual instances of these types. (If the -v display option is used at the same time, all symbols will be displayed -- not just those with associated type information.) You can also replace *Name* with a period (.) to signify that you want to repeat the most recently used value of *Name*.

If *Name* contains a space, it should be enclosed in parentheses.

Field

Specifies the field(s) to be displayed. If *Field* is omitted, all fields are displayed. If *Field* is followed by a period (.), the first-level subfields of this field will be displayed as well. If *Field* is followed with a series of periods, the subfields will be displayed to a depth equal

to the number of periods. Any field name followed by a period will be treated as a prefix match, as if the **-y** search option was used. If *Field* is followed by an asterisk (*), it is treated as only the beginning of the field, not necessarily the entire field, and all matching fields are displayed.

Address

Specifies the address of the structure to be displayed. If *Name* is omitted, *Address* must be included and must specify the address of a global variable. *Address* is taken to be a virtual address unless otherwise specified. Use the **-p** option to specify a physical address. Use an "at" sign (@) to specify a register (for example, @eax).

List

Specifies the field name that links a linked list. The *Address* parameter must be included.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The **dt** command output will always display signed numbers in base 10, and unsigned numbers in hexadecimal.

All parameters of **dt** that allow symbol values also allow string wildcards. See [String Wildcard Syntax](#) for details.

The **-y** and **-n** options can precede any *Name* or *Field*. The **-y** option allows you to specify the beginning of the type or structure name. For example, **dt -y ALLEN** will display data about the type **ALLEN**TOWN. However, you could not display the type **ALLEN**TOWN with **dt -y A**. Instead, you would have to use **dt -ny A**, because **A** is a valid hexadecimal value and would be interpreted as an address without the **-n** option.

If *Name* indicates a structure, all fields will be displayed (for example, **dt myStruct**). If you only want one specific field, you can do **dt myStruct myField**. This displays the member that C would call **myStruct.myField**. However, note that the command **dt myStruct myField1 myField2** displays **myStruct.myField1** and **myStruct.myField2**. It does not display **myStruct.myField1.myField2**.

If a structure name or field is followed by a subscript, this specifies a single instance of an array. For example, **dt myStruct myFieldArray[3]** will display the fourth element of the array in question. But if a type name is followed by a subscript, this specifies an entire array. For example, **dt CHAR[8] myPtr** will display an eight-character string. The subscript is always taken as decimal regardless of the current radix; an **0x** prefix will cause an error.

Because the command uses type information from the *.pdb* file, it can freely be used to debug any CPU platform.

The type information used by **dt** includes all type names created with **typedef**, including all the Windows-defined types. For example, **unsigned long** and **char** are not valid type names, but **ULONG** and **CHAR** are. See the Microsoft Windows SDK for a full list of all Windows type names.

All types created by **typedefs** within your own code will be present, as long as they have actually been used in your program. However, types that are defined in your headers but never actually used will not be stored in the *.pdb* symbol files and will not be accessible to the debugger. To make such a type available to the debugger, use it as the *input* of a **typedef** statement. For example, if the following appears in your code, the structure **MY_DATA** will be stored in the *.pdb* symbol file and can be displayed by the **dt** command:

```
dbgcmd

typedef struct _MY_DATA {
    ...
} MY_DATA;
typedef MY_DATA *PMY_DATA;
```

On the other hand, the following code would not suffice because both **MY_DATA** and **PMY_DATA** are defined by the initial **typedef** and, therefore, **MY_DATA** has not itself been used as the input of any **typedef** statement:

```
dbgcmd

typedef struct _MY_DATA {
    ...
}
```

```
} MY_DATA, *PMY_DATA;
```

In any event, type information is included only in a full symbol file, not a symbol file that has been stripped of all private symbol information. For more information, see [Public and Private Symbols](#).

If you want to display unicode strings, you need to use the [.enable_unicode \(Enable Unicode Display\)](#) command first. You can control the display of long integers with the [.enable_long_status \(Enable Long Integer Display\)](#) command.

In the following example, **dt** displays a global variable:

```
dbgcmd

0:000> dt mt1
+0x000 a : 10
+0x004 b : 98 'b'
+0x006 c : 0xdd
+0x008 d : 0xabcd
+0x00c gn : [6] 0x1
+0x024 ex : 0x0
```

In the following example, **dt** displays the array field **gn**:

```
dbgcmd

0:000> dt mt1 -a gn
+0x00c gn :
[00] 0x1
[01] 0x2
[02] 0x3
[03] 0x4
[04] 0x5
[05] 0x6
```

The following command displays some subfields of a variable:

```
dbgcmd

0:000> dt mcl1 m_t1 dpo
+0x010 dpo : DEEP_ONE
+0x070 m_t1 : MYTYPE1
```

The following command displays the subfields of the field **m_t1**. Because the period automatically causes prefix matching, this will also display subfields of any field that begins with "m_t1":

```
dbgcmd
```

```
0:000> dt mcl1 m_t1.  
+0x070 m_t1 :  
+0x000 a : 0  
+0x004 b : 0 '  
+0x006 c : 0x0  
+0x008 d : 0x0  
+0x00c gn : [6] 0x0  
+0x024 ex : 0x0
```

You could repeat this to any depth. For example, the command **dt mcl1 a..c** would display all fields to depth four, such that the first field name began with a and the third field name began with c.

Here is a more detailed example of how subfields can be displayed. First, display the **Ldr** field:

```
dbgcmd
```

```
0:000> dt nt!_PEB Ldr 7ffd000  
+0x00c Ldr : 0x00191ea0
```

Now expand the pointer type field:

```
dbgcmd
```

```
0:000> dt nt!_PEB Ldr Ldr. 7ffd000  
+0x00c Ldr : 0x00191ea0  
+0x000 Length : 0x28  
+0x004 Initialized : 0x1 '  
+0x008 SsHandle : (null)  
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x191ee0 - 0x192848 ]  
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x191ee8 - 0x192850 ]  
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x191f58 -  
0x192858 ]  
+0x024 EntryInProgress : (null)
```

Now display the **CriticalSectionTimeout** field:

```
dbgcmd
```

```
0:000> dt nt!_PEB CriticalSectionTimeout 7ffd000  
+0x070 CriticalSectionTimeout : _LARGE_INTEGER 0xfffffe86d`079b8000
```

Now expand the **CriticalSectionTimeout** structure subfields one level deep:

```
dbgcmd
```

```
0:000> dt nt!_PEB CriticalSectionTimeout. 7ffd000
+0x070 CriticalSectionTimeout : 0xfffffe86d`079b8000
+0x000 LowPart : 0x79b8000
+0x004 HighPart : -6035
+0x000 u : __unnamed
+0x000 QuadPart : -2592000000000000
```

Now expand the **CriticalSectionTimeout** structure subfields two levels deep:

```
dbgcmd
```

```
0:000> dt nt!_PEB CriticalSectionTimeout.. 7ffd000
+0x070 CriticalSectionTimeout : 0xfffffe86d`079b8000
+0x000 LowPart : 0x79b8000
+0x004 HighPart : -6035
+0x000 u :
+0x000 LowPart : 0x79b8000
+0x004 HighPart : -6035
+0x000 QuadPart : -2592000000000000
```

The following command displays an instance of the data type MYTYPE1 that is located at the address 0x0100297C:

```
dbgcmd
```

```
0:000> dt 0x0100297c MYTYPE1
+0x000 a : 22
+0x004 b : 43 '+'
+0x006 c : 0x0
+0x008 d : 0x0
+0x00c gn : [6] 0x0
+0x024 ex : 0x0
```

The following command displays an array of 10 ULONGs at the address 0x01002BE0:

```
dbgcmd
```

```
0:000> dt -ca10 ULONG 01002be0
[0] 0x1001098
[1] 0x1
[2] 0xdead
[3] 0x7d0
[4] 0x1
[5] 0xcd
[6] 0x0
[7] 0x0
```

```
[8] 0x0  
[9] 0x0
```

The following command continues the previous display at a different address. Note that "ULONG" does not need to be re-entered:

```
dbgcmd  
  
0:000> dt -ca4 . 01002d00  
Using sym ULONG  
[0] 0x12  
[1] 0x4ac  
[2] 0xbadfeed  
[3] 0x2
```

Here are some examples of type display. The following command displays all types and globals beginning with the string "MY" in the module *thismodule*. Those prefixed with an address are actual instances; those without addresses are type definitions:

```
dbgcmd  
  
0:000> dt thismodule!MY*  
010029b8 thismodule!myglobal1  
01002990 thismodule!myglobal2  
thismodule!MYCLASS1  
thismodule!MYCLASS2  
thismodule!MYCLASS3  
thismodule!MYTYPE3::u  
thismodule!MYTYPE1  
thismodule!MYTYPE3  
thismodule!MYTYPE3  
thismodule!MYFLAGS
```

When performing type display, the **-v** option can be used to display the size of each item. The **-s size** option can be used to only enumerate items of a specific size. Again, those prefixed with an address are actual instances; those without addresses are type definitions:

```
dbgcmd  
  
0:001> dt -s 2 -v thismodule!*  
Enumerating symbols matching thismodule!*, Size = 0x2  
Address  Size Symbol  
002 thismodule!wchar_t  
002 thismodule!WORD  
002 thismodule!USHORT  
002 thismodule!SHORT  
002 thismodule!u_short
```

```
002 thismodule!WCHAR
00427a34 002 thismodule!numberOfShips
00427a32 002 thismodule!numberOfPlanes
00427a30 002 thismodule!totalNumberOfItems
```

Here is an example of the **-b** option. The structure is expanded and the **OwnerThreads** array within the structure is expanded, but the **Flink** and **Blink** list pointers are not followed:

```
dbgcmd
```

```
kd> dt nt!_ERESOURCE -b 0x8154f040
+0x000 SystemResourcesList : [ 0x815bb388 - 0x816cd478 ]
    +0x000 Flink      : 0x815bb388
    +0x004 Blink      : 0x816cd478
    +0x008 OwnerTable : (null)
    +0x00c ActiveCount : 1
    +0x00e Flag       : 8
    +0x010 SharedWaiters : (null)
    +0x014 ExclusiveWaiters : (null)
    +0x018 OwnerThreads :
        [00]
            +0x000 OwnerThread   : 0
            +0x004 OwnerCount   : 0
            +0x004 TableSize     : 0
        [01]
            +0x000 OwnerThread   : 0x8167f563
            +0x004 OwnerCount   : 1
            +0x004 TableSize     : 1
    +0x028 ContentionCount : 0
    +0x02c NumberOfSharedWaiters : 0
    +0x02e NumberOfExclusiveWaiters : 0
    +0x030 Address       : (null)
    +0x030 CreatorBackTraceIndex : 0
    +0x034 SpinLock      : 0
```

Here is an example of **dt** in kernel mode. The following command produces results similar to [!process 0 0](#):

```
dbgcmd
```

```
kd> dt nt!_EPROCESS -l ActiveProcessLinks.Flink -y Ima -yoI Uni 814856f0
## ActiveProcessLinks.Flink at 0x814856f0

UniqueProcessId : 0x00000008
ImageFileName : [16] "System"

## ActiveProcessLinks.Flink at 0x8138a030

UniqueProcessId : 0x00000084
ImageFileName : [16] "smss.exe"
```

```
## ActiveProcessLinks.Flink at 0x81372368

UniqueProcessId : 0x000000a0
ImageFileName : [16] "csrss.exe"

## ActiveProcessLinks.Flink at 0x81369930

UniqueProcessId : 0x000000b4
ImageFileName : [16] "winlogon.exe"

....
```

If you want to execute a command for each element of the list, use the [!list](#) extension.

Finally, the **dt -h** command will display a short help text summarizing the **dt** syntax.

dtx (Display Type - Extended Debugger Object Model Information)

Article • 10/25/2023

The dtx command displays extended symbolic type information using the debugger object model. The dtx command is similar to the [dt \(Display Type\) command](#).

dbgcmd

```
dtx -DisplayOpts [Module!]Name Address
```

Parameters

DisplayOpts

Use the following optional flags to change how the output is displayed.

-a Displays array elements in a new line with its index.

-r [n] Recursively dump the subtypes (fields) up to *n* levels.

-h Displays command line help for the dtx command.

Module!

An optional parameter specifying the module that defines this structure, followed by the exclamation point. If there is a local variable or type with the same name as a global variable or type, you should include *module* name to specify the global variable.

Name

A type name or a global symbol.

Address

Memory address containing the type.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The following examples show how to use the dtx command.

Use the address and the name to display extended symbolic type information.

```
dbgcmd
```

```
0: kd> dtx nt!_EPROCESS fffffb607560b56c0
(*((nt!_EPROCESS *)0xfffffb607560b56c0)) [Type: _EPROCESS]
    [+0x000] Pcb [Type: _KPROCESS]
    [+0x2d8] ProcessLock [Type: _EX_PUSH_LOCK]
    [+0x2e0] RundownProtect [Type: _EX_RUNDOWN_REF]
    [+0x2e8] UniqueProcessId : 0x4 [Type: void *]
    [+0x2f0] ActiveProcessLinks [Type: _LIST_ENTRY]
```

Display additional information using the -r recursion option.

```
dbgcmd
```

```
0: kd> dtx -r2 HdAudio!CAzMixerTopoMiniport ffffff806`d24992b8
(*((HdAudio!CAzMixerTopoMiniport *)0xfffff806d24992b8))
[Type: CAzMixerTopoMiniport]
    [+0x018] m_lRefCount : -766760880 [Type: long]
    [+0x020] m_pUnknownOuter : 0xfffff806d24dbc40 [Type: IUnknown *]
    [+0x028] m_FilterDesc [Type: PCFILTER_DESCRIPTOR]
        [+0x000] Version : 0xd24c2890 [Type: unsigned long]
        [+0x008] AutomationTable : 0xfffff806d24c2780 [Type:
PCAUTOMATION_TABLE *]
            [+0x000] PropertyItemSize : 0x245c8948 [Type: unsigned long]
            [+0x004] PropertyCount : 0x6c894808 [Type: unsigned long]
            [+0x008] Properties : 0x5718247489481024 [Type:
PCPROPERTY_ITEM *]
                [+0x010] MethodItemSize : 0x55415441 [Type: unsigned long]
                [+0x014] MethodCount : 0x57415641 [Type: unsigned long]
                [+0x018] Methods : 0x4ce4334540ec8348 [Type:
PCMETHOD_ITEM *]
                    [+0x020] EventItemSize : 0x8b41f18b [Type: unsigned long]
                    [+0x024] EventCount : 0xd8b48f4 [Type: unsigned long]
                    [+0x028] Events : 0x7d2d8d4cfffd854 [Type:
PCEVENT_ITEM *]
                        [+0x030] Reserved : 0x66ffffd79 [Type: unsigned long]
                        [+0x010] PinSize : 0xd24aa9b0 [Type: unsigned long]
                        [+0x014] PinCount : 0xfffff806 [Type: unsigned long]
                        [+0x018] Pins : 0xfffff806d24aa740 [Type:
PCPIN_DESCRIPTOR *]
                            [+0x000] MaxGlobalInstanceCount : 0x57555340 [Type: unsigned
long]
                            [+0x004] MaxFilterInstanceCount : 0x83485741 [Type: unsigned
long]
                            [+0x008] MinFilterInstanceCount : 0x8b4848ec [Type: unsigned
long]
```

```
[+0x010] AutomationTable  : 0xa5158b48ed33c000 [Type:  
PCAUTOMATION_TABLE *]  
[+0x018] KsPinDescriptor  [Type: KSPIN_DESCRIPTOR]
```

Tip: Use the [x \(Examine Symbols\)](#) command to display the address of an item of interest.

```
dbgcmd
```

```
0: kd> x /d HdAudio!CazMixertopoMiniport*  
...  
fffff806`d24992b8 HdAudio!CAzMixertopoMiniport::`vftable' = <no type  
information>  
...
```

See also

[dt \(Display Type\)](#)

dv (Display Local Variables)

Article • 10/25/2023

The **dv** command displays the names and values of all local variables in the current scope.

```
dbgcmd
```

```
dv [Flags] [Pattern]
```

Parameters

Flags

Causes additional information to be displayed. Any of the following case-sensitive *Flags* can be included:

/f <addr>

Lets you specify an arbitrary function address so that you can see what parameters and locals exist for any code anywhere. It turns off the value display and implies **/V**. The **/f** flag must be the last flag. A parameter filter pattern can still be specified after it if the string is quoted.

/i

Causes the display to specify the kind of variable: local, global, parameter, function, or unknown.

/t

Causes the display to include the data type for each local variable.

/v

Causes the display to include the virtual memory address or register location of each local variable.

/V

Same as **/v**, and also includes the address of the local variable relative to the relevant register.

/a

Sorts the output by address, in ascending order.

/A

Sorts the output by address, in descending order.

/n

Sorts the output by name, in ascending order.

/N

Sorts the output by name, in descending order.

/z

Sorts the output by size, in ascending order.

/Z

Sorts the output by size, in descending order.

Pattern

Causes the command to only display local variables that match the specified *Pattern*.

The pattern may contain a variety of wildcards and specifiers; see [String Wildcard Syntax](#) for details. If *Pattern* contains spaces, it must be enclosed in quotation marks. If *Pattern* is omitted, all local variables will be displayed.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details on displaying and changing local variables and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

In verbose mode, the addresses of the variables are displayed as well. (This can also be done with the [x \(Examine Symbols\)](#) command.)

Data structures and unfamiliar data types are not displayed in full; rather, their type name is displayed. To display the entire structure, or display a particular member of the structure, use the [dt \(Display Type\)](#) command.

The *local context* determines which set of local variables will be displayed. By default, this context matches the current position of the program counter. For information about how this can be changed, see [Local Context](#).

dx (Display Debugger Object Model Expression)

Article • 10/25/2023

The **dx** command displays a C++ expression using the NatVis extension model. For more information about NatVis, see [Create custom views of native objects](#).

dbgcmd

```
dx [-g|-gc #][-c #][-n|-v]-r[#] Expression[,<FormatSpecifier> ]  
dx [{-?}|{-h}]
```

Parameters

Expression

A C++ expression to be displayed.

-g

Display as a data grid objects which are iterable. Each iterated element is a row in the grid and each display child of those elements is a column. This allows you to view something such as an array of structs, where each array element is displayed in a row and each field of the struct is displayed in a column.

Selecting a column name (where there is an available DML link) will sort by that column. If already sorted by that column, the sort order will be inverted.

Any object which is iterable will have a select and hold (or right-click) context menu item added via DML called 'Display as Grid'. Selecting and holding (or right-clicking) an object in the output window and selecting this will display the object in the grid view instead of the standard tree view.

A (+) displayed by a column name offers both a select-and-hold (or right-click) and a select behavior.

- Select takes that column and explodes it into its own table. You see the original rows plus the children of the expanded column.
- Select and hold (or right-click) provides "Expand Into Grid" which takes the column and adds it back to the current table as right most columns.

-gc #

Display as a grid and restrict grid cell sizes to specified number of (#) characters.

-c # Displays container continuation (skipping # elements of the container). This option is typically used in custom output automation scenarios and provides a "..." continuation element at the bottom of the listing.

-n There are two ways that data can be rendered. Using the NatVis visualization (the default) or using the underlying native C/C++ structures. Specify the -n parameter to render the output using just the native C/C++ structures and not the NatVis visualizations.

-v

Display verbose information that includes methods and other non-typical objects.

-r#

Recursively display subtypes (fields) up to # levels. If # is not specified, a recursion level of one, is the default value.

[<FormatSpecifier>]

Use any of the following format specifiers to modify the default rendering.

Format specifier	Description
,x	Display ordinals in hexadecimal
,d	Display ordinals in decimal
,o	Display ordinals in octal
,b	Display ordinals in binary
,en	Display enums by name only (no value)
,c	Display as single character (not a string)
,s	Display 8-bit strings as ASCII quoted
,sb	Display 8-bit strings as ASCII unquoted
,s8	Display 8-bit strings as UTF-8 quoted
,s8b	Display 8-bit strings as UTF-8 unquoted
,su	Display 16-bit strings as UTF-16 quoted
,sub	Display 16-bit strings as UTF-16 unquoted

Format specifier	Description
,!	Display objects in raw mode only (e.g. no NatVis)
,#	Specify length of pointer/array/container as the literal value # (replace with numeric)
,[<expression>]	Specify length of pointer/array/container as the expression <expression>
,nd	Do not find the derived (runtype) type of the object. Display static value only

dx -?

Display command line help.

dx -h Displays help for objects available in the debugger.

dx -id

Microsoft internal use only. Used to follow data model links in command output.

Command line usage example

The .dx settings command can be used to display information about the Debug Settings object. For more information about the debug settings objects, see [.settings](#).

```
dbgcmd

kd> dx -r1 Debugger.Settings
Debugger.Settings
  Debug
  Display
  EngineInitialization
  Extensions
  Input
  Sources
  Symbols
  AutoSaveSettings : false
```

Use the -r1 recursion option to view the other Debugger objects - Sessions, Settings and State.

```
dbgcmd

kd> dx -r1 Debugger
Debugger
  Sessions
```

Settings
State
Utility
LastEvent

Specify the Debugger.Sessions object with the -r3 recursion option to travel further down the object chain.

dbgcmd

```
kd> dx -r3 Debugger.Sessions
Debugger.Sessions
[0] : Remote KD:
KdSrv:Server=@{<Local>},Trans=@{1394:Channel=0}
Processes
[0] : <Unknown Image>
[4] : <Unknown Image>
[304] : smss.exe
[388] : csrss.exe
[456] : wininit.exe
[468] : csrss.exe
[528] : services.exe
[536] : lsass.exe
[544] : winlogon.exe
[620] : svchost.exe
...
...
```

Add the x format specifier to display the ordinal values in hexadecimal.

dbgcmd

```
kd> dx -r3 Debugger.Sessions,x
Debugger.Sessions,x
[0x0] : Remote KD:
KdSrv:Server=@{<Local>},Trans=@{1394:Channel=0}
Processes
[0x0] : <Unknown Image>
[0x4] : <Unknown Image>
[0x130] : smss.exe
[0x184] : csrss.exe
[0x1c8] : wininit.exe
[0x1d4] : csrss.exe
[0x210] : services.exe
[0x218] : lsass.exe
[0x220] : winlogon.exe
[0x26c] : svchost.exe
[0x298] : svchost.exe
[0x308] : dwm.exe
[0x34c] : nvvsvc.exe
[0x37c] : nvvsvc.exe
```

```
[0x384] : svchost.exe  
...     ...
```

This example uses an active debug session to list the call stack of the first thread in the first process.

dbgcmd

```
kd> dx -r1  
Debugger.Sessions.First().Processes.First().Threads.First().Stack.Frames  
Debugger.Sessions.First().Processes.First().Threads.First().Stack.Frames  
[0x0]      : nt!RtlpBreakWithStatusInstruction  
[0x1]      : nt!KdCheckForDebugBreak + 0x7a006  
[0x2]      : nt!KiUpdateRunTime + 0x42  
[0x3]      : nt!KiUpdateTime + 0x129  
[0x4]      : nt!KeClockInterruptNotify + 0x1c3  
[0x5]      : hal!HalpTimerClockInterruptEpilogCommon + 0xa  
[0x6]      : hal!HalpTimerClockInterruptCommon + 0x3e  
[0x7]      : hal!HalpTimerClockInterrupt + 0x1cb  
[0x8]      : nt!KiIdleLoop + 0x1a
```

Use the **-g** option to display output as a data grid. Select column to sort.

dbgcmd

```
kd> dx -g @$curprocess.Modules
```

BaseAddress	Name	Size
0xfffff8017ca0000	\SystemRoot\System32\drivers\ksecdd.sys	0x26000
0xfffff8017ca30000	\SystemRoot\System32\drivers\clipesp.sys	0x4d000
0xfffff8017ca80000	\SystemRoot\System32\drivers\cmncontext.sys	0xe000
0xfffff8017ca90000	\SystemRoot\System32\drivers\ntosext.sys	0xc000
0xfffff8017caaa000	\SystemRoot\system32\CL.dll	0x96000
0xfffff8017cb40000	\SystemRoot\system32\drivers\Wdf01000.sys	0xdd000
0xfffff8017cc10000	\SystemRoot\system32\drivers\WDFLDR.SYS	0x12000
0xfffff8017cc40000	\SystemRoot\system32\Drivers\Wdf01000.sys	0x12000
0xfffff8017cc70000	\SystemRoot\System32\Drivers\VppRecorder.sys	0x1000
0xfffff8017cc80000	\SystemRoot\System32\drivers\ACPI.sys	0x94000
0xfffff8017cd20000	\SystemRoot\System32\drivers\WMILIB.SYS	0xc000
0xfffff8017cd30000	\SystemRoot\System32\Drivers\cng.sys	0x94000
0xfffff8017cd80000	\SystemRoot\System32\drivers\pcv.sys	0x12000
0xfffff8017ce00000	\SystemRoot\System32\drivers\msisadrv.sys	0x8000
0xfffff8017ce10000	\SystemRoot\System32\drivers\pci.sys	0x4f000
0xfffff8017ce80000	\SystemRoot\System32\drivers\vdvroot.sys	0xf000
0xfffff8017ce70000	\SystemRoot\System32\drivers\pdc.sys	0x1b000

Use the **-h** option to display information about objects.

dbgcmd

```
kd> dx -h Debugger.State  
Debugger.State [State pertaining to the current execution of the debugger  
(e.g.: user variables)]  
    DebuggerVariables [Debugger variables which are owned by the debugger  
and can be referenced by a pseudo-register prefix of @$]  
    PseudoRegisters [Categorized debugger managed pseudo-registers which  
can be referenced by a pseudo-register prefix of @$]  
    UserVariables [User variables which are maintained by the debugger  
and can be referenced by a pseudo-register prefix of @$]
```

Displaying TEB and PEB information using the Environment object

Use the Environment object to display TEB and PEB information associated with the thread and process.

To display the TEB associated with the current thread use this command.

```
dbgcmd

0: kd> dx -r2 @$curthread.Environment
@$curthread.Environment
    EnvironmentBlock [Type: _TEB]
        [+0x000] NtTib           [Type: _NT_TIB]
        [+0x038] EnvironmentPointer : Unable to read memory at Address 0x38
        [+0x040] ClientId         [Type: _CLIENT_ID]
        [+0x050] ActiveRpcHandle   : Unable to read memory at Address 0x50
        [+0x058] ThreadLocalStoragePointer : Unable to read memory at
Address 0x58
        [+0x060] ProcessEnvironmentBlock : Unable to read memory at Address
0x60
        [+0x068] LastErrorValue   : Unable to read memory at Address 0x68
        [+0x06c] CountOfOwnedCriticalSection : Unable to read memory at
Address 0x6c
        [+0x070] CsrClientThread   : Unable to read memory at Address 0x70
        [+0x078] Win32ThreadInfo   : Unable to read memory at Address 0x78
        [+0x080] User32Reserved    [Type: unsigned long [26]]
        [+0x0e8] UserReserved      [Type: unsigned long [5]]
        [+0x100] WOW32Reserved     : Unable to read memory at Address 0x100
        [+0x108] CurrentLocale     : Unable to read memory at Address 0x108
        [+0x10c] FpSoftwareStatusRegister : Unable to read memory at Address
0x10c
        ...
...
```

To display PEB associated with the current process use this command.

```
dbgcmd

0: kd> dx -r2 @$curprocess.Environment
@$curprocess.Environment
    EnvironmentBlock [Type: _PEB]
        [+0x000] InheritedAddressSpace : Unable to read memory at Address
0x0
        [+0x001] ReadImageFileExecOptions : Unable to read memory at Address
0x1
        [+0x002] BeingDebugged      : Unable to read memory at Address 0x2
        [+0x003] BitField            : Unable to read memory at Address 0x3
        [+0x003 ( 0: 0)] ImageUsesLargePages : Unable to read memory at
Address 0x3
        [+0x003 ( 1: 1)] IsProtectedProcess : Unable to read memory at
```

```
Address 0x3
    [+0x003 ( 2: 2)] IsImageDynamicallyRelocated : Unable to read memory
at Address 0x3
    [+0x003 ( 3: 3)] SkipPatchingUser32Forwarders : Unable to read
memory at Address 0x3
    [+0x003 ( 4: 4)] IsPackagedProcess : Unable to read memory at
Address 0x3
    [+0x003 ( 5: 5)] IsAppContainer : Unable to read memory at Address
0x3
    [+0x003 ( 6: 6)] IsProtectedProcessLight : Unable to read memory at
Address 0x3
    [+0x003 ( 7: 7)] IsLongPathAwareProcess : Unable to read memory at
Address 0x3
    [+0x004] Padding0 [Type: unsigned char [4]]
    [+0x008] Mutant : Unable to read memory at Address 0x8
    [+0x010] ImageBaseAddress : Unable to read memory at Address 0x10
    [+0x018] Ldr : Unable to read memory at Address 0x18
    [+0x020] ProcessParameters : Unable to read memory at Address 0x20
...
...
```

Kernel Io.Handles object

Use the current process Io.Handles object to display kernel handle information.

```
dbgcmd

0: kd> dx -r1 @$curprocess.Io.Handles
@$curprocess.Io.Handles
[0x8]
[0xc]
[0x10]
[0x14]
[0x18]
...
...
```

Use the .First() function to display information about the first handle.

```
dbgcmd

0: kd> dx -r2 @$curprocess.Io.Handles.First()
@$curprocess.Io.Handles.First()
Handle : 0x8
Type : Unexpected failure to dereference object
GrantedAccess : Unexpected failure to dereference object
Object [Type: _OBJECT_HEADER]
    [+0x000] PointerCount : 228806 [Type: __int64]
    [+0x008] HandleCount : 6 [Type: __int64]
    [+0x008] NextToFree : 0x6 [Type: void *]
    [+0x010] Lock [Type: _EX_PUSH_LOCK]
    [+0x018] TypeIndex : 0xf2 [Type: unsigned char]
```

```

[+0x019] TraceFlags      : 0x0 [Type: unsigned char]
[+0x019 ( 0: 0)] DbgRefTrace    : 0x0 [Type: unsigned char]
[+0x019 ( 1: 1)] DbgTracePermanent : 0x0 [Type: unsigned char]
[+0x01a] InfoMask       : 0x0 [Type: unsigned char]
[+0x01b] Flags          : 0x2 [Type: unsigned char]
[+0x01b ( 0: 0)] NewObject     : 0x0 [Type: unsigned char]
[+0x01b ( 1: 1)] KernelObject   : 0x1 [Type: unsigned char]
[+0x01b ( 2: 2)] KernelOnlyAccess : 0x0 [Type: unsigned char]
[+0x01b ( 3: 3)] ExclusiveObject : 0x0 [Type: unsigned char]
[+0x01b ( 4: 4)] PermanentObject : 0x0 [Type: unsigned char]
[+0x01b ( 5: 5)] DefaultSecurityQuota : 0x0 [Type: unsigned char]
[+0x01b ( 6: 6)] SingleHandleEntry : 0x0 [Type: unsigned char]
[+0x01b ( 7: 7)] DeletedInline   : 0x0 [Type: unsigned char]
[+0x01c] Reserved        : 0x0 [Type: unsigned long]
[+0x020] ObjectCreateInfo : 0xfffffff801f6d9c6c0 [Type:
_OBJECT_CREATE_INFORMATION *]
[+0x020] QuotaBlockCharged : 0xfffffff801f6d9c6c0 [Type: void *]
[+0x028] SecurityDescriptor : 0xfffffb984aa815d06 [Type: void *]
[+0x030] Body             [Type: _QUAD]
ObjectType      : Unexpected failure to dereference object
UnderlyingObject : Unexpected failure to dereference object

```

Note that the Io.Handles object is a kernel only object.

Working around symbol file limitations with casting

When displaying information about various Windows system variables, there are times where not all of the type information is available in the public symbols. This example illustrates this situation.

```

dbgcmd

0: kd> dx nt!PsIdleProcess
Error: No type (or void) for object at Address 0xfffffff800e1d50128

```

The dx command supports the ability to reference the address of a variable which does not have type information. Such “address of” references are treated as “void *” and can be cast as such. This means that if the data type is known, the following syntax can be used to display type information for the variable.

```

dbgcmd

dx (Datatype *)&VariableName

```

For example for a nt!PsIdleProcess which has a data type of nt!_EPROCESS, use this command.

```
dbgcmd

dx (nt!_EPROCESS *)&nt!PsIdleProcess
(nt!_EPROCESS *)&nt!PsIdleProcess : 0xfffff800e1d50128
[Type: _EPROCESS *]
[+0x000] Pcb [Type: _KPROCESS]
[+0x2c8] ProcessLock [Type: _EX_PUSH_LOCK]
[+0x2d0] CreateTime : {4160749568} [Type: _LARGE_INTEGER]
[+0x2d8] RundownProtect [Type: _EX_RUNDOWN_REF]
[+0x2e0] UniqueProcessId : 0x1000 [Type: void *]
[+0x2e8] ActiveProcessLinks [Type: _LIST_ENTRY]
[+0x2f8] Flags2 : 0x218230 [Type: unsigned long]
[+0x2f8 ( 0: 0)] JobNotReallyActive : 0x0 [Type: unsigned long]
```

The dx command does not support switching expression evaluators with the @@ MASM syntax. For more information about expression evaluators, see [Evaluating Expressions](#).

Using LINQ With the debugger objects

LINQ syntax can be used with the debugger objects to search and manipulate data. LINQ is conceptually similar to the Structured Query Language (SQL) that is used to query databases. You can use a number of LINQ methods to search, filter and parse debug data. For information us using LINQ with the debugger objects, see [Using LINQ With the debugger objects](#).

Using debugger objects with NatVis and JavaScript

For information about using debugger objects with NatVis, see [Native Debugger Objects in NatVis](#).

For information about using debugger objects with JavaScript, see [Native Debugger Objects in JavaScript Extensions](#).

See also

[Using LINQ With the debugger objects](#)

[Native Debugger Objects in NatVis](#)

Native Debugger Objects in JavaScript Extensions

e, ea, eb, ed, eD, ef, ep, eq, eu, ew, eza (Enter Values)

Article • 03/06/2024

The `*e*` commands enter into memory the values that you specify.

This command should not be confused with the [~E \(Thread-Specific Command\)](#) qualifier.

```
dbgcmd
e{b|d|D|f|p|q|w} Address [Values]
e{a|u|za|zu} Address "String"
e Address [Values]
```

Kernel Mode only - physical address

```
dbgcmd
/p {[c]| [uc] | [wc]}
```

For more information, see [Kernel mode physical addresses](#) later in this topic.

Parameters

Syntax

Address

Specifies the starting address where to enter values. The debugger replaces the value at *Address* and each subsequent memory location until all *Values* have been used.

Values

Specifies one or more values to enter into memory. Multiple numeric values should be separated with spaces. If you do not specify any values, the current address and the value at that address will be displayed, and you will be prompted for input.

String

Specifies a string to be entered into memory. The **ea** and **eza** commands will write this to memory as an ASCII string; the **eu** and **ezu** commands will write this to memory as a

Unicode string. The **eza** and **ezu** commands write a terminal NULL; the **ea** and **eu** commands do not. *String* must be enclosed in quotation marks.

Environment

 Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

This command exists in the following forms. The second characters of the **ed** and **eD** commands are case-sensitive.

 Expand table

Command	Enter
e	This enters data in the same format as the most recent e command. (If the most recent e command was ea , eza , eu , or ezu , the final parameter will be <i>String</i> and may not be omitted.)
ea	ASCII string (not NULL-terminated).
eb	Byte values.
ed	Double-word values (4 bytes).
eD	Double-precision floating-point numbers (8 bytes).
ef	Single-precision floating-point numbers (4

Command	Enter
	bytes).
ep	Pointer-sized values. This command is equivalent to ed or eq , depending on whether the target computer's processor architecture is 32-bit or 64-bit, respectively.
eq	Quad-word values (8 bytes).
eu	Unicode string (not NULL-terminated).
ew	Word values (2 bytes).
eza	NULL-terminated ASCII string.
ezu	NULL-terminated Unicode string.

Numeric values will be interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the [n \(Set Number Base\)](#) command. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Note The default radix behaves differently when C++ expressions are being used. See [Evaluating Expressions](#) for details.

When entering byte values with the **eb** command, you can use single straight quotation marks to specify characters. If you want to include multiple characters, each must be surrounded with its own quotation marks. This allows you to enter a character string that is not terminated by a null character. For example:

```
dbgcmd
eb 'h' 'e' 'l' 'l' 'o'
```

C-style escape characters (such as `\\\0` or `\\\n`) may not be used with these commands.

If you omit the *Values* parameter, you will be prompted for input. The address and its current contents will be displayed, and an **Input>** prompt will appear. You can then do any of the following:

- Enter a new value, by typing the value and pressing ENTER.
- Preserve the current value in memory by pressing SPACE followed by ENTER.
- Exit from the command by pressing ENTER.

Kernel mode physical addresses

In WinDbg version 1.2402.24001.0 and later, the `e` (Enter Memory) commands supports physical addresses just like `d`, `da`, `db`, `dc`, `dd`, `dD`, `df`, `dp`, `dq`, `du`, `dw`, `dW`, `dyb`, `dyd` ([Display Memory](#)). These options are only supported in kernel mode.

[] [Expand table](#)

Option	Description
<code>/p</code>	Uses physical memory addresses for the display. The range specified by Range will be taken from physical memory rather than virtual memory.
<code>/p[c]</code>	Same as <code>/p</code> , except that cached memory will be read. The brackets around c must be included.
<code>/p[uc]</code>	Same as <code>/p</code> , except that uncached memory will be read. The brackets around uc must be included.
<code>/p[wc]</code>	Same as <code>/p</code> , except that write-combined memory will be read. The brackets around wc must be included.

See also

[Reading and Writing Memory](#)

[d, da, db, dc, dd, dD, df, dp, dq, du, dw, dW, dyb, dyd \(Display Memory\)](#)

[Evaluating Expressions](#)

f, fp (Fill Memory)

Article • 10/25/2023

The **f** and **fp** commands fill the specified memory range with a repeating pattern.

These commands should not be confused with the [~F \(Freeze Thread\)](#) command.

```
dbgcmd  
f Range Pattern  
fp [MemoryType] PhysicalRange Pattern
```

Parameters

Range

Specifies the range in virtual memory to fill. For more syntax details, see [Address and Address Range Syntax](#).

PhysicalRange

(Kernel mode only) Specifies the range in physical memory to fill. The syntax of *PhysicalRange* is the same as that of a virtual memory range, except that no symbol names are permitted.

MemoryType

(Kernel mode only) Specifies the type of physical memory, which can be one of the following:

[c]

Cached memory.

[uc]

Uncached memory.

[wc]

Write-combined memory.

Pattern

Specifies one or more byte values with which to fill memory.

Environment

Modes

	f : user mode, kernel mode fp : kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

This command fills the memory area specified by *range* with the specified *pattern*, repeated as many times as necessary.

The *pattern* parameter must be input as a series of bytes. These can be entered as numeric or as ASCII characters.

Numeric values will be interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the [n \(Set Number Base\)](#) command. The default radix can be overridden by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

Note The default radix behaves differently when C++ expressions are being used. For more information, see the [Evaluating Expressions](#) topic.

If ASCII characters are used, each character must be enclosed in single straight quotation marks. C-style escape characters (such as '\0' or '\n') may not be used.

If multiple bytes are specified, they must be separated by spaces.

If *pattern* has more values than the number of bytes in the range, the debugger ignores the extra values.

Here are some examples. Assuming the current radix is 16, the following command will fill memory locations 0012FF40 through 0012FF5F with the pattern "ABC", repeated several times:

```
dbgcmd
0:000> f 0012ff40 L20 'A' 'B' 'C'
```

The following command has the exact same effect:

```
dbgcmd
```

```
0:000> f 0012ff40 L20 41 42 43
```

The following examples show how you can use the physical memory types (**c**, **uc**, and **wc**) with the **fp** command in kernel mode:

```
dbgcmd
```

```
kd> fp [c] 0012ff40 L20 'A' 'B' 'C'
```

```
dbgcmd
```

```
kd> fp [uc] 0012ff40 L20 'A' 'B' 'C'
```

```
dbgcmd
```

```
kd> fp [wc] 0012ff40 L20 'A' 'B' 'C'
```

g (Go)

Article • 10/25/2023

The **g** command starts executing the given process or thread. Execution will halt at the end of the program, when *BreakAddress* is hit, or when another event causes the debugger to stop.

User-Mode Syntax

```
dbgcmd  
[~Thread] g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
dbgcmd  
g[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread

(User mode only) Specifies the thread to execute. For syntax details, see [Thread Syntax](#).

a

Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress

Specifies the address where execution should begin. If this is not specified, the debugger passes execution to the address specified by the current value of the instruction pointer. For more syntax details, see [Address and Address Range Syntax](#).

BreakAddress

Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress*

cannot be resolved, it is stored as an [unresolved breakpoint](#). For more syntax details, see [Address and Address Range Syntax](#).

BreakCommands

Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the **g** command, and any additional commands listed after the semicolon will be executed immediately after the **g** command is done.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see [Controlling the Target](#).

Remarks

If *Thread* is specified, then the **g** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123g**, **~#g**, or **~*g** command is specified, the specified threads are unfrozen and all others are frozen.

See also

[gu \(Go Up\)](#)

[gh \(Go with Exception Handled\)](#)

gc (Go from Conditional Breakpoint)

Article • 10/25/2023

The `gc` command resumes execution from a conditional breakpoint in the same fashion that was used to hit the breakpoint (stepping, tracing, or freely executing). This only applies to the older style of conditional breakpoints using a "j (Condition) ..." style expression, rather than the simpler "/w" style conditional breakpoint. For more information, see [setting a conditional breakpoint](#).

```
dbgcmd
```

```
gc
```

While this command is no longer as useful for conditional breakpoints, it can still be used for breakpoints that do logging or some other activity without breaking into the debugger. For instance, a breakpoint could be written that looks like this:

```
dbgcmd
```

```
bp module!myFunction ".echo myFunction executed; gc"
```

If a normal "g" command were used instead, the program would continue execution when stepping over "myFunction", instead of simply printing the message and continuing the step operation.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For an overview of related commands, see [Controlling the Target](#).

Remarks

When a [conditional breakpoint](#) using a "j (Condition) ..." expression includes an execution command at the end, this should be the **gc** command.

For example, the following is an example conditional breakpoint:

```
dbgcmd
```

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'gc' "
```

When this breakpoint is encountered and the expression is false, execution will resume using the same execution type that was previously used. For example, if you used a **g** (**Go**) command to reach this breakpoint, execution would resume freely. But if you reached this breakpoint while stepping or tracing, execution would resume with a step or a trace.

On the other hand, the following is an improper breakpoint formulation, since execution will always resume freely even if you had been stepping before reaching the breakpoint:

```
dbgcmd
```

```
0:000> bp Address "j (Condition) 'OptionalCommands'; 'g' "
```

gh (Go with Exception Handled)

Article • 10/25/2023

The **gh** command marks the given thread's exception as having been handled and allows the thread to restart execution at the instruction that caused the exception.

User-Mode Syntax

```
dbgcmd
```

```
[~Thread] gh[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
dbgcmd
```

```
gh[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread

(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see [Thread Syntax](#).

a

Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress

Specifies the address at which execution should begin. If this is not specified, the debugger passes execution to the address where the exception occurred. For more syntax details, see [Address and Address Range Syntax](#).

BreakAddress

Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress* cannot be resolved, it is stored as an [unresolved breakpoint](#). For more syntax details, see [Address and Address Range Syntax](#).

BreakCommands

Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the **gh** command, and any additional commands listed after the semicolon will be executed immediately after the **gh** command is done.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see [Controlling the Target](#).

Remarks

If you use the *BreakAddress* parameter to set a breakpoint, this new breakpoint will only be triggered by the current thread. Other threads that execute the code at that location will not be stopped.

If *Thread* is specified, then the **gh** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gh**, **~#gh**, or **~*gh** command is specified, the specified threads are unfrozen and all others are frozen.

gn, gN (Go with Exception Not Handled)

Article • 10/25/2023

The **gn** and **gN** commands continue execution of the given thread without marking the exception as having been handled. This allows the application's exception handler to handle the exception.

User-Mode Syntax

```
dbgcmd
```

```
[~Thread] gn[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]  
[~Thread] gN[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Kernel-Mode Syntax

```
dbgcmd
```

```
gn[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]  
gN[a] [= StartAddress] [BreakAddress ... [; BreakCommands]]
```

Parameters

Thread

(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see [Thread Syntax](#).

a

Causes any breakpoint created by this command to be a processor breakpoint (like those created by **ba**) rather than a software breakpoint (like those created by **bp** and **bm**). If *BreakAddress* is not specified, no breakpoint is created and the **a** flag has no effect.

StartAddress

Specifies the address where execution should begin. If this is not specified, the debugger passes execution to the address where the exception occurred. For more syntax details, see [Address and Address Range Syntax](#).

BreakAddress

Specifies the address for a breakpoint. If *BreakAddress* is specified, it must specify an instruction address (that is, the address must contain the first byte of an instruction). Up to ten break addresses, in any order, can be specified at one time. If *BreakAddress*

cannot be resolved, it is stored as an [unresolved breakpoint](#). For more syntax details, see [Address and Address Range Syntax](#).

BreakCommands

Specifies one or more commands to be automatically executed when the breakpoint specified by *BreakAddress* is hit. The *BreakCommands* parameter must be preceded by a semicolon. If multiple *BreakAddress* values are specified, *BreakCommands* applies to all of them.

Note The *BreakCommands* parameter is only available when you are embedding this command within a command string used by another command -- for example, within another breakpoint command or within an except or event setting. On a command line, the semicolon will terminate the command, and any additional commands listed after the semicolon will be executed immediately after the **gn** or **gN** command is done.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see [Controlling the Target](#).

Remarks

If the debugger is not stopped at a breakpoint, **gn** and **gN** behave identically. If the debugger is stopped at a breakpoint, **gn** will not work; you must capitalize the "N" to execute this command. This is a safety precaution, since it is rarely wise to continue a breakpoint unhandled.

If you use the *BreakAddress* parameter to set a breakpoint, this new breakpoint will only be triggered by the current thread. Other threads that execute the code at that location will not be stopped.

If *Thread* is specified, then the **gn** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gn**, **~#gn**, or **~*gn** command is

specified, the specified threads are unfrozen and all others are frozen.

gu (Go Up)

Article • 10/25/2023

The **gu** command causes the target to execute until the current function is complete.

User-Mode Syntax

```
dbgcmd  
[~Thread] gu
```

Kernel-Mode Syntax

```
dbgcmd  
gu
```

Parameters

Thread

(User mode only) Specifies the thread to execute. This thread must have been stopped by an exception. For syntax details, see [Thread Syntax](#).

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For other methods of issuing this command and an overview of related commands, see [Controlling the Target](#).

Remarks

The **gu** command executes the target until the current function call returns.

If the current function is called recursively, the **gu** command will not halt execution until the *current instance* of the current function returns. In this way, **gu** differs from **g @\$ra**, which will halt any time the return address of this function is hit.

Note The **gu** command distinguishes different instances of a function by measuring the call stack depth. Executing this command in assembly mode after the arguments have been pushed to the stack and just before the call is made may cause this measurement to be incorrect. Function returns that are optimized away by the compiler may similarly cause this command to stop at the wrong instance of this return. These errors are rare, and can only happen during recursive function calls.

If *Thread* is specified, then the **gu** command is executed with the specified thread unfrozen and all others frozen. For example, if the **~123gu**, **~#gu**, or **~*gu** command is specified, the specified threads are unfrozen and all others are frozen.

ib, iw, id (Input from Port)

Article • 10/25/2023

The **ib**, **iw**, and **id** commands read and display a byte, word, or double word from the selected port.

dbgcmd

```
ib Address
iw Address
id Address
```

Parameters

Address

The address of the port.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based computer only

Remarks

The **ib** command reads a single byte, the **iw** command reads a word, and the **id** command reads a double word.

Make sure that reading an I/O port does not affect the behavior of the device that you are reading from. Some devices change state after a read-only port has been read. You should also not try to read a word or double-word from a port that does not allow values of this length.

See also

[ob, od, ow \(Output to Port\)](#)

j (Execute If - Else)

Article • 10/25/2023

The `j` command conditionally executes one of the specified commands, depending on the evaluation of a given expression.

dbgcmd

```
j Expression Command1 ; Command2  
j Expression 'Command1' ; 'Command2'
```

Parameters

Expression

The expression to evaluate. If this expression evaluates to a nonzero value, *Command1* is executed. If this expression evaluates to zero, *Command2* is executed. For more information about the syntax of this expression, see [Numerical Expression Syntax](#).

Command1

The command string to be executed if the expression in *Expression* evaluates to a nonzero value (TRUE). You can combine multiple commands by surrounding the command string with single straight quotation marks (') and separating commands by using semicolons. If the command string is a single command, the single quotation marks are optional.

Command2

The command string to be executed if the expression in *Expression* evaluates to zero (FALSE). You can combine multiple commands by surrounding the command string with single straight quotation marks (') and separating commands by using semicolons. If the command string is a single command, the single quotation marks are optional.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You cannot add a semicolon or additional commands after the **j** command. If a semicolon appears after *Command2*, everything after the semicolon is ignored.

The following command displays the value of **eax** if **MySymbol** is equal to zero and displays the values of **ebx** and **ecx** otherwise.

```
dbgcmd  
0:000> j (MySymbol=0) 'r eax'; 'r ebx; r ecx'
```

You could omit the single quotation marks around **r eax**, but they make the command easier to read. If you want to omit one of the commands, you can include empty quotation marks or omit the parameter for that command, as in the following commands.

```
dbgcmd  
0:000> j (MySymbol=0) ''; 'r ebx; r ecx'  
0:000> j (MySymbol=0) ; 'r ebx; r ecx'
```

You can also use the **j** command inside other commands. For example, you can use a **j** command to create conditional breakpoints.

```
dbgcmd  
0:000> bp `mysource.cpp:143` "j (poi(MyVar)>0n20) ''; 'gc' "
```

This style of conditional breakpoint is no longer recommended, as a simpler form is now available in the debugger. For more information about the syntax of conditional breakpoints, see [Setting a Conditional Breakpoint](#).

See also

[z \(Execute While\)](#)

k, kb, kc, kd, kp, kP, kv (Display Stack Backtrace)

Article • 10/25/2023

The **k*** commands display the stack frame of the given thread with related information.

User-Mode, x86 Processor

```
dbgcmd

[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr StackPtr InstructionPtr
[~Thread] kd [WordCount]
```

Kernel-Mode, x86 Processor

```
dbgcmd

[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = BasePtr StackPtr InstructionPtr
[Processor] kd [WordCount]
```

User-Mode, x64 Processor

```
dbgcmd

[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr
FrameCount
[~Thread] kd [WordCount]
```

Kernel-Mode, x64 Processor

```
dbgcmd

[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr
FrameCount
[Processor] kd [WordCount]
```

User-Mode, ARM Processor

```
dbgcmd
```

```
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[~Thread] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr
FrameCount
[~Thread] kd [WordCount]
```

Kernel-Mode, ARM Processor

```
dbgcmd
```

```
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] [FrameCount]
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr FrameCount
[Processor] k[b|p|P|v] [c] [n] [f] [L] [M] = StackPtr InstructionPtr
FrameCount
[Processor] kd [WordCount]
```

Parameters

Thread

Specifies the thread stack to be displayed. If you omit this parameter, the stack of the current thread is displayed. For more information about thread syntax, see [Thread syntax](#). You can specify threads only in user mode.

Processor

Specifies the processor whose stack is to be displayed. For more information about processor syntax, see [Multiprocessor Syntax](#).

b

Displays the first three parameters that are passed to each function in the stack trace.

c

Displays a clean stack trace. Each display line includes only the module name and the function name.

p

Displays all of the parameters for each function that's called in the stack trace. The parameter list includes each parameter's data type, name, and value. The p option is case sensitive. This parameter requires full symbol information.

P

Displays all of the parameters for each function that's called in the stack trace, like the `p` parameter. However, for `P`, the function parameters are printed on a second line of the display, instead of on the same line as the rest of the data.

V

Displays frame pointer omission (FPO) information. On x86-based processors, the display also includes calling convention information.

N

Displays frame numbers.

F

Displays the distance between adjacent frames. This distance is the number of bytes that separate the frames on the actual stack.

L

Hides source lines in the display. `L` is case sensitive.

M

Displays the output using [Debugger markup language](#). Each frame number in the display is a link that you can select to set the local context and display local variables. For information about the local context, see [.frame](#).

FrameCount

Specifies the number of stack frames to display. You should specify this number in hexadecimal format unless you've changed the radix by using the [n \(set number base\)](#) command. Use the [.kframes \(set stack length\)](#) command to display the default and to change the value.

BasePtr

Specifies the base pointer for the stack trace. The `BasePtr` parameter is available only if there's an equal sign (=) after the command.

StackPtr

Specifies the stack pointer for the stack trace. If you omit `StackPtr` and `InstructionPtr`, the command uses the stack pointer that the `rsp` (or `esp`) register specifies and the instruction pointer that the `rip` (or `eip`) register specifies.

InstructionPtr

Specifies the instruction pointer for the stack trace. If you omit `StackPtr` and `InstructionPtr`, the command uses the stack pointer that the `rsp` (or `esp`) register specifies and the instruction pointer that the `rip` (or `eip`) register specifies.

WordCount

Specifies the number of DWORD_PTR values in the stack to dump.

Environment	
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When you issue the `k`, `kb`, `kp`, `kP`, or `kv` commands, a stack trace is displayed in a tabular format. If line loading is enabled, source modules and line numbers are also displayed.

The stack trace includes the base pointer for the stack frame, the return address, and the function names.

If you use the `kp` or `kP` commands, the full parameters for each function that's called in the stack trace are displayed. The parameter list includes each parameter's data type, name, and value.

The command might be slow. For example, when `MyFunction1` calls `MyFunction2`, the debugger must have full symbol information for `MyFunction1` to display the parameters that are passed in this call. This command doesn't fully display internal Microsoft Windows routines that aren't exposed in public symbols.

If you use the `kb` or `kv` commands, the first three parameters that are passed to each function are displayed. If you use the `kv` command, FPO data is also displayed.

On an x86-based processor, the `kv` command also displays calling convention information.

When you use the `kv` command, the FPO information is added at the end of the line in the following format.

FPO text	Meaning
FPO: [non-Fpo]	No FPO data for the frame.
FPO: [N1,N2,N3]	<p><i>N1</i> is the total number of parameters.</p> <p><i>N2</i> is the number of DWORD values for the local variables.</p>

FPO text	Meaning
	N_3 is the number of registers that are saved.
FPO: [N1,N2] TrapFrame @ Address	N_1 is the total number of parameters. N_2 is the number of DWORD values for the locals. <i>Address</i> is the address of the trap frame.
FPO: TaskGate Segment:0	<i>Segment</i> is the segment selector for the task gate.
FPO: [EBP 0xBase]	<i>Base</i> is the base pointer for the frame.

The `kd` command displays the raw stack data. Each DWORD value is displayed on a separate line. Symbol information is displayed for those lines together with associated symbols. This format creates a more detailed list than the other `k*` commands. The `kd` command is equivalent to a [dds \(display memory\)](#) command that uses the stack address as its parameter.

If you use the `k` command at the beginning of a function (before the function prolog has been executed), you receive incorrect results. The debugger uses the frame register to compute the current backtrace, and this register isn't set correctly for a function until its prolog has been executed.

In user mode, the stack trace is based on the stack of the current thread. For more information about threads, see [Controlling processes and threads](#).

In kernel mode, the stack trace is based on the current [register context](#). You can set the register context to match a specific thread, context record, or trap frame.

Additional information

For more information about the Register context and other context settings, see [Changing contexts](#).

I+, I- (Set Source Options)

Article • 10/25/2023

The **I+** and **I-** commands set the source line options that control source display and program stepping options.

dbgcmd

1+Option
1-Option
1{+|-}

Parameters

+ or -

Specifies whether a given option is to be turned on (plus sign [+]) or turned off (minus sign [-]).

Option

One of the following options. The options must be in lowercase letters.

I

Displays source line numbers at the command prompt. You can disable source line display through **I-ls** or **.prompt_allow -src**. To make the source line numbers visible, you must enable source line display through both mechanisms.

o

Hides all messages (other than the source line and line number) when you are stepping through code. (The **s** option must also be active for the **o** option to have any effect.)

s

Displays source lines and source line numbers at the command prompt.

t

Starts [source mode](#). If this mode is not set, the debugger is in [assembly mode](#).

Turns on or turns off all options.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about source debugging and related commands, see [Debugging in Source Mode](#). For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

If you omit *Option*, the previously set options are displayed. In this case, the **I+** and **I-** commands have identical effects. However, you must include a plus sign (+) or minus sign (-) for the **I** command to work.

You can include only one *Option* every time that you issue this command. If you list more than one option, only the first option is detected. However, by repeatedly issuing this command, you can turn on or off as many options as you want. (In other words, **I+Ist** does not work, but **I+I; I+s; I+t** does achieve the effect that you want.)

When you specify the **s** option, source lines and line numbers are displayed when you step through code, regardless of whether you specified the **I** option. The **o** option has no effect unless you specify the **s** option.

Source line options do not take effect unless you enable line number loading by using the [.lines \(Toggle Source Line Support\)](#) command or the [-lines command-line option](#). By default, if you have not used these commands, WinDbg turns on source line support and CDB turns it off.

!ld (Load Symbols)

Article • 10/25/2023

The **!ld** command loads symbols for the specified module and updates all module information.

dbgcmd

```
!ld ModuleName [/f FileName]
```

Parameters

ModuleName

Specifies the name of the module whose symbols are to be loaded. *ModuleName* can contain a variety of wildcard characters and specifiers.

/f FileName

Changes the name selected for the match. By default the module name is matched, but when */f* is used the file name is matched instead of the module name. *FileName* can contain a variety of wildcard characters and specifiers. For more information on the syntax of wildcard characters and specifiers, see [String Wildcard Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The debugger's default behavior is to use *lazy symbol loading* (also known as [deferred symbol loading](#)). This means that symbols are not actually loaded until they are needed.

The **!ld** command, on the other hand, forces all symbols for the specified module to be loaded.

Additional Information

For more information about deferred (lazy) symbol loading, see [Deferred Symbol Loading](#). For more information about other symbol options, see [Setting Symbol Options](#).

!m (List Loaded Modules)

Article • 10/25/2023

The **!m** command displays the specified loaded modules. The output includes the status and the path of the module.

```
dbgcmd
```

```
!m Options [a Address] [m Pattern | M Pattern]
```

Parameters

Options

Any combination of the following options:

D

Displays output using [Debugger Markup Language](#).

O

Displays only loaded modules.

I

Displays only modules whose symbol information has been loaded.

V

Causes the display to be verbose. The display includes the symbol file name, the image file name, checksum information, version information, date stamps, time stamps, and information about whether the module is managed code (CLR). This information is not displayed if the relevant headers are missing or paged out.

U

(Kernel mode only) Displays only user-mode symbol information.

K

(Kernel mode only) Displays only kernel-mode symbol information.

E

Displays only modules that have a symbol problem. These symbols include modules that have no symbols and modules whose symbol status is C, T, #, M, or Export. For more information about these notations, see [Symbol Status Abbreviations](#).

c

Displays checksum data.

1m

Reduces the output so that nothing is included except the names of the modules. This option is useful if you are using the [.foreach](#) token to pipe the command output into another command's input.

sm

Sorts the display by module name instead of by the start address.

In addition, you can include only one of the following options. If you do not include any of these options, the display includes the symbol file name.

i

Displays the image file name.

f

Displays the full image path. (This path always matches the path that is displayed in the initial load notification, unless you issued a [.reload -s](#) command.) When you use f, symbol type information is not displayed.

n

Displays the image name. When you use n, symbol type information is not displayed.

p

Displays the mapped image name. When you use p, symbol type information is not displayed.

t

Displays the file time stamps. When you use t, symbol type information is not displayed.

a *Address*

Specifies an address that is contained in this module. Only the module that contains this address is displayed. If Address contains an expression, it must be enclosed in parentheses.

m Pattern

Specifies a pattern that the module name must match. Pattern can contain a variety of wildcard characters and specifiers. For more information about the syntax of this information, see [String Wildcard Syntax](#).

In most cases, the module name is the file name without the file name extension. For example, if you want to display information about the Flpydisk.sys driver, use the `!m flpydisk` command, not `!m mflydisk.sys`. In some cases, the module name differs significantly from the file name.

M Pattern

Specifies a pattern that the image path must match. Pattern can contain a variety of wildcard characters and specifiers. For more information about the syntax of this information, see [String Wildcard Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The `!m` command lists all of the modules and the status of symbols for each module.

Windows maintains an unloaded module list for user-mode processes. When you are debugging a user-mode process or dump file, the `!m` command also shows these unloaded modules.

The modules displayed depends on how you are debugging, for example user or kernel mode, and the specific context you are looking at. For more information about the process context and other context settings, see [Changing Contexts and Controlling Processes and Threads](#).

This command shows several columns or fields, each with a different title. Some of these titles have specific meanings:

- *module name* is typically the file name without the file name extension. In some cases, the module name differs significantly from the file name.
- The symbol type immediately follows the module name. This column is not labeled. For more information about the various status values, see [Symbol Status Abbreviations](#). If you have loaded symbols, the symbol file name follows this column.
- The first address in the module is shown as start. The first address after the end of the module is shown as end. For example, if start is "faab4000" and end is "faab8000", the module extends from 0xFAAB4000 to 0xFAAB7FFF, inclusive.
- **!mv only:** The image path column shows the name of the executable file, including the file name extension. Typically, the full path is included in user mode but not in kernel mode.
- **!mv only:** The loaded symbol image file value is the same as the image name, unless Microsoft CodeView symbols are present.
- **!mv only:** The mapped memory image file value is typically not used. If the debugger is mapping an image file (for example, during minidump debugging), this value is the name of the mapped image.

The following code example shows the **!m** command using the **m** and **s*** options, so only modules that begin with "s" are displayed.

```
dbgcmd

kd> !m m s*
start      end      module name
f9f73000  f9f7fd80  sysaudio    (deferred)
fa04b000  fa09b400  srv         (deferred)
faab7000  faac8500  sr          (deferred)
facac000  facbae00  serial      (deferred)
fb008000  fb00ba80  serenum     e:\mysymbols\SereEnum.pdb\.....
fb24f000  fb250000  swenum      (deferred)

Unloaded modules:
f9f53000  f9f61000  swmidi.sys
fb0ae000  fb0b0000  splitter.sys
fb040000  fb043000  Sfloppy.SYS
```

Examples

The following two examples show the **!m** command once without any options and once with the **sm** option. Compare the sort order in the two examples.

Example 1:

```
dbgcmd

0:000> lm
start   end     module name
01000000 0100d000  stst      (deferred)
77c10000 77c68000  msrvct    (deferred)
77dd0000 77e6b000  ADVAPI32  (deferred)
77e70000 77f01000  RPCRT4   (deferred)
7c800000 7c8f4000  kernel32  (deferred)
7c900000 7c9b0000  ntdll     (private pdb symbols) c:\db20sym\ntdll.pdb
```

Example 2:

```
dbgcmd

0:000> lm sm
start   end     module name
77dd0000 77e6b000  ADVAPI32  (deferred)
7c800000 7c8f4000  kernel32  (deferred)
77c10000 77c68000  msrvct    (deferred)
7c900000 7c9b0000  ntdll     (private pdb symbols) c:\db20sym\ntdll.pdb
77e70000 77f01000  RPCRT4   (deferred)
01000000 0100d000  stst      (deferred)
```

See also

- [Changing Contexts](#)
- [Controlling Processes and Threads.](#)
- [Standard debugging techniques](#)
- [Get started with Windows Debugging](#)

In (List Nearest Symbols)

Article • 10/25/2023

The **In** command displays the symbols at or near the given address.

dbgcmd
In Address
In /D Address

Parameters

Address

Specifies the address where the debugger should start to search for symbols. The nearest symbols, either before or after *Address*, are displayed. For more information about the syntax, see [Address and Address Range Syntax](#).

/D

Specifies that the output is displayed using [Debugger Markup Language \(DML\)](#). The DML output includes a link that you can use to explore the module that contains the nearest symbol. It also includes a link that you can use to set a breakpoint.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You can use the **In** command to help determine what a pointer is pointing to. This command can also be useful when you are looking at a corrupted stack to determine which procedure made a call.

If source line information is available, the **In** display also includes the source file name and line number information.

If you are using a [source server](#), the **In** command displays information that is related to the source server.

Is, Isa (List Source Lines)

Article • 10/25/2023

The **Is** and **Isa** commands display a series of lines from the current source file and advance the current source line number.

dbgcmd

```
ls [.] [first] [, count]  
isa [.] address [, first [, count]]
```

Parameters

.
Causes the command to look for the source file that the debugger engine or the [.srcpath \(Set Source Path\)](#) command are using. If the period (.) is not included, **Is** uses the file that was most recently loaded with the [!sf \(Load Source File\)](#) command.

address

Specifies the address where the source display is to begin.

For more information about the syntax, see [Address and Address Range Syntax](#).

first

Specifies the first line to display. The default value is the current line.

count

Specifies the quantity of lines to display. The default value is 20 (0x14), unless you have changed the default value by using the [!sp -a](#) command.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

After you run the **Is** or **Isa** command, the current line is redefined as the final line that is displayed plus one. The current line is used in future **Is**, **Isa**, and **Isc** commands.

See also

[Isc \(List Current Source\)](#)

[Isf, Isf- \(Load or Unload Source File\)](#)

lsc (List Current Source)

Article • 10/25/2023

The **lsc** command displays the current source file name and line number.

```
dbgcmd
```

```
lsc
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

[ls, lsa \(List Source Lines\)](#)

[lsf, lsf- \(Load or Unload Source File\)](#)

Ise (Launch Source Editor)

Article • 10/25/2023

The **Ise** command opens an editor for the current source file.

```
dbgcmd
```

```
Ise
```

Environment

Modes	User-mode, kernel-mode
Targets	Live, crash dump
Platforms	All

Remarks

The **Ise** command opens an editor for the current source file. This command is equivalent to clicking **Edit this file** in the shortcut menu of the [Source window](#) in WinDbg.

The editor is opened on the computer that the target is running on, so you cannot use the **Ise** command from a remote client.

The WinDiff editor registry information or the value of the `WINDBG_INVOKE_EDITOR` environment variable determine which editor is opened. For example, consider the following value of `WINDBG_INVOKE_EDITOR`.

```
reg
```

```
c:\my\path\myeditor.exe -file %f -line %l
```

This value indicates that Myeditor.exe opens to the one-based line number of the current source file. The `%l` option indicates that line numbers should be read as one-based, and `%f` indicates that the current source file should be used. You could also include `%L` to indicate that line numbers are zero-based or `%p` to indicate that the current source file should be used.

!sf, !sf- (Load or Unload Source File)

Article • 10/25/2023

The **!sf** and **!sf-** commands load or unload a source file.

dbgcmd

```
!sf Filename  
!sf- Filename
```

Parameters

Filename

Specifies the file to load or unload. If this file is not located in the directory where the debugger was opened from, you must include an absolute or relative path. The file name must follow Microsoft Windows file name conventions.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **!sf** command loads a source file.

The **!sf-** command unloads a source file. You can use this command to unload files that you previously loaded with **!sf** or automatically loaded source files. You cannot use **!sf-** to unload files that were loaded through WinDbg's **File | Open Source File** command or files that a WinDbg workspace loaded.

In CDB or KD, you can view source files in the [Debugger Command window](#). In WinDbg, source files are loaded as new [Source windows](#).

For more information about source files, source paths, and other ways to load source files, see [Source Path](#).

See also

[ls, lsa \(List Source Lines\)](#)

[lsc \(List Current Source\)](#)

Isp (Set Number of Source Lines)

Article • 10/25/2023

The **Isp** command controls how many source lines are displayed while you step through or execute code or use the [Is and Isa commands](#).

dbgcmd

```
lsp [-a] LeadingLines TrailingLines  
lsp [-a] TotalLines  
lsp [-a]
```

Parameters

-a

Sets or displays the number of lines that **Is** and **Isa** show. If you omit **-a**, **Isp** sets or displays the number of lines that are shown while you step through and execute code.

LeadingLines

Specifies the number of lines to show before the current line.

TrailingLines

Specifies the number of lines to show after the current line.

TotalLines

Specifies the total number of lines to show. This number is divided evenly between leading and trailing lines. (If this number is odd, more trailing lines are displayed.)

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When you use the **Isp** command together with no parameters, **Isp** displays the current leading line and trailing line values that you used while stepping. When you use this command together with only the **-a** parameter, **Isp** displays the values that you used while stepping and for the [Is and Isa commands](#).

When you step through a program or break in after program execution, the previous **Isp** command determines the number of leading and trailing lines that are displayed. When you use **Isa**, the previous **Isp -a** command determines the number of leading and trailing lines that are displayed. When you use **Is**, all lines appear as a single block, so the previous **Isp -a** command determines the total number of lines that are displayed.

Additional Information

For more information about source debugging and related commands, see [Debugging in Source Mode](#).

m (Move Memory)

Article • 10/25/2023

The **m** command copies the contents of memory from one location to another.

Do not confuse this command with the [~m \(Resume Thread\)](#) command.

dbgcmd
m Range Address

Parameters

Range

Specifies the memory area to copy. For more information about the syntax of this parameter, see [Address and Address Range Syntax](#).

Address

Specifies the starting address of the destination memory area. For more information about the syntax of this parameter, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The memory area that *Address* specifies can be part of the memory area that *Range* specifies. Overlapping moves are handled correctly.

n (Set Number Base)

Article • 10/25/2023

The **n** command sets the default number base (radix) to the specified value or displays the current number base.

Do not confuse this command with the [~n \(Suspend Thread\)](#) command.

dbgcmd
n [Radix]

Parameters

Radix

Specifies the default number base that is used for numeric display and entry. You can use one of the following values.

Value	Description
8	Octal
10	Decimal
16	Hexadecimal

If you omit *Radix*, the current default number base is displayed.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The current radix affects the input and output of MASM expressions. It does not affect the input or output of C++ expressions. For more information about these expressions,

see [Evaluating Expressions](#).

The default radix is set to 16 when the debugger is started.

In all MASM expressions, numeric values are interpreted as numbers in the current radix (16, 10, or 8). You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary).

ob, ow, od (Output to Port)

Article • 10/25/2023

The **ob**, **ow**, and **od** commands send a byte, word, or double word to the selected port.

dbgcmd

```
ob Address Value
ow Address Value
od Address Value
```

Parameters

Address

Specifies the address of the port.

Value

Specifies the hexadecimal value to write to the port.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based only

Remarks

The **ob** command writes a single byte, the **ow** command writes a word, and the **od** command writes a double word.

Make sure that you do not send a word or a double-word to a port that does not support this size.

See also

[ib, id, iw \(Input from Port\)](#)

p (Step)

Article • 10/25/2023

The **p** command executes a single instruction or source line and optionally displays the resulting values of all registers and flags. When subroutine calls or interrupts occur, they are treated as a single step.

User-Mode

```
dbgcmd
```

```
[~Thread] p[r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
dbgcmd
```

```
p[r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread

Specifies the threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **pr**, **tr**, or **.prompt_allow -reg** commands. All three of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where execution should begin. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of instructions or source lines to step through before stopping. Each step is displayed as a separate action in the [Debugger Command window](#). The default value is one.

Command

Specifies a debugger command to execute after the step is performed. This command is executed before the standard **p** results are displayed. If you also use *Count*, the specified command is executed after all stepping is complete (but before the results from the final step are displayed).

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about issuing the **p** command and an overview of related commands, see [Controlling the Target](#).

Remarks

When you specify *Count*, each instruction is displayed as it is stepped through.

If the debugger encounters a **call** instruction or interrupt while stepping, the called subroutine will execute completely unless a breakpoint is encountered. Control is returned to the debugger at the next instruction after the call or interrupt.

Each step executes a single assembly instruction or a single source line, depending on whether the debugger is in assembly mode or source mode. Use the **I+t** and **I-t** commands or the buttons on the WinDbg toolbar to switch between these modes.

When you are quickly stepping many times in WinDbg, the debugging information windows are updated after each step. If this update causes slower response time, use [.suspend_ui \(Suspend WinDbg Interface\)](#) to temporarily suspend the refreshing of these windows.

pa (Step to Address)

Article • 10/25/2023

The **pa** command executes the program until the specified address is reached, displaying each step.

User-Mode

dbgcmd

```
[~Thread] pa [r] [= StartAddress] StopAddress ["Command"]
```

Kernel-Mode

dbgcmd

```
pa [r] [= StartAddress] StopAddress ["Command"]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **par**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress

Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

StopAddress

Specifies the address where execution will stop. This address must match the exact address of an instruction.

Command

Specifies a debugger command to execute after the step is performed. This command is executed before the standard **pa** results are displayed. If you also use *StopAddress*, the specified command is executed after *StopAddress* is reached (but before the results from the final step are displayed).

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **pa** command causes the target to begin executing. This execution continues until the specified instruction is reached or a breakpoint is encountered.

Note If you use this command in kernel mode, execution stops when an instruction is encountered at the specified virtual address in any virtual address space.

During this execution, all steps are displayed explicitly. Called functions are treated as a single unit. Therefore, the display of this command is similar to what you see if you execute **p (Step)** repeatedly until the program counter reaches the specified address.

For example, the following command explicitly steps through the target code until the return address of the current function is reached.

```
dbgcmd
```

```
0:000> pa @$ra
```

The following example demonstrates using the **pa** command along with the **kb** command to display the stack trace:

```
dbgcmd  
0:000> pa 70b5d2f1 "kb"
```

pc (Step to Next Call)

Article • 10/25/2023

The **pc** command executes the program until a call instruction is reached.

User-Mode

```
dbgcmd  
[~Thread] pc [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
pc [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **pcr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **call** instructions that the debugger must encounter for this

command to stop. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **pc** command causes the target to begin executing. This execution continues until a **call** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **call** instruction, the entire call is executed. After this call is returned, execution continues until another **call** is reached. This execution, rather than tracing, of the call is the only difference between **pc** and [tc \(Trace to Next Call\)](#).

In source mode, you can associate one source line with multiple assembly instructions. The **pc** command does not stop at a **call** instruction that is associated with the current source line.

pct (Step to Next Call or Return)

Article • 10/25/2023

The **pct** command executes the program until it reaches a call instruction or a return instruction.

User-Mode

```
dbgcmd  
[~Thread] pct [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
pct [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display through the **pctr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress

Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **call** or **return** instructions that must be encountered for this command to stop. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **pct** command causes the target to begin executing. This execution continues until a **call** or **return** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **call** or **return** instruction, the entire call or return is executed. After this call or return is returned, execution continues until another **call** or **return** is reached. This execution, rather than tracing, of the call is the only difference between **pct** and [tct \(Trace to Next Call or Return\)](#).

In source mode, you can associate one source line with multiple assembly instructions. The **pct** command does not stop at a **call** or **return** instruction that is associated with the current source line.

ph (Step to Next Branching Instruction)

Article • 10/25/2023

The **ph** command executes the program until any kind of branching instruction is reached, including conditional or unconditional branches, calls, returns, and system calls.

User-Mode

```
dbgcmd  
[~Thread] ph [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
ph [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **phr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the [**rm \(Register Mask\)**](#) command.

StartAddress

Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of branching instructions that must be encountered for this command to stop. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **ph** command causes the target to begin executing. This execution continues until a branching instruction is reached or a breakpoint is encountered.

If the program counter is already on a branching instruction, the entire branching instruction is executed. After this branching instruction is returned, execution continues until another branching instruction is reached. This execution, rather than tracing, of the call is the only difference between **ph** and [th \(Trace to Next Branching Instruction\)](#).

In source mode, you can associate one source line with multiple assembly instructions. The **ph** command does not stop at a branching instruction that is associated with the current source line.

pt (Step to Next Return)

Article • 10/25/2023

The **pt** command executes the program until a return instruction is reached.

User-Mode

```
dbgcmd  
[~Thread] pt [r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
dbgcmd  
pt [r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **ptr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the [**rm \(Register Mask\)**](#) command.

StartAddress

Specifies the address where the debugger begins execution. Otherwise, the debugger begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **return** instructions that must be encountered for this command

to stop. The default value is one.

Command

Specifies a debugger command to execute after the step is performed. This command is executed before the standard **pt** results are displayed. If you also use *Count*, the specified command is executed after all stepping is complete (but before the results from the final step are displayed).

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **pt** command causes the target to begin executing. This execution continues until a **return** instruction is reached or a breakpoint is encountered.

If the program counter is already on a **return** instruction, the entire return is executed. After this return is returned, execution continues until another **return** is reached. This execution, rather than tracing, of the call is the only difference between **pt** and [tt \(Trace to Next Return\)](#).

In source mode, you can associate one source line with multiple assembly instructions. The **pt** command does not stop at a **return** instruction that is associated with the current source line.

The following example demonstrates using the **pt** command along with the **kb** command to display the stack trace:

```
dbgcmd
```

```
0:000> pt "kb"
```

q, qq (Quit)

Article • 10/25/2023

The **q** and **qq** commands end the debugging session. (In CDB and KD, this command also exits the debugger itself. In WinDbg, this command returns the debugger to dormant mode.)

```
dbgcmd

q
qq
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In user-mode debugging, the **q** command ends the debugging session and closes the target application.

In kernel-mode debugging, the **q** command saves the log file and ends the debugging session. The target computer remains locked.

If this command does not work in KD, press **CTRL+R+ENTER** on the debugger keyboard, and then retry the **q** command. If this action does not work, you must use **CTRL+B+ENTER** to exit the debugger.

The **qq** command behaves exactly like the **q** command, unless you are performing remote debugging. During remote debugging, the **q** command has no effect, but the **qq** command ends the debugging server. For more information about this effect, see [Remote Debugging Through the Debugger](#).

qd (Quit and Detach)

Article • 10/25/2023

The **qd** command ends the debugging session and leaves any user-mode target application running. (In CDB and KD, this command also exits the debugger itself. In WinDbg, this command returns the debugger to dormant mode.)

dbgcmd
qd

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

The **qd** command detaches from a target application and ends the debugging session, leaving the target still running. However, this command is supported only on Microsoft Windows XP and later versions of Windows. On Windows 2000, **qd** generates a warning message and has no effect.

When you are performing remote debugging through the debugger, you cannot use the **qd** command from a debugging client.

r (Registers)

Article • 10/25/2023

The **r** command displays or modifies registers, floating-point registers, flags, pseudo-registers, and fixed-name aliases.

User-Mode

```
dbgcmd
```

```
[~Thread] r[M Mask|F|X|?] [ Register[:[Num]Type] [= [Value]] ]  
r.
```

Kernel-Mode

```
dbgcmd
```

```
[Processor] r[M Mask|F|X|Y|YI|?] [ Register[:[Num]Type] [= [Value]] ]  
r.
```

Parameters

Processor

Specifies the processor that the registers are read from. The default value is zero. If you specify *Processor*, you cannot include the *Register* parameter--all registers are displayed. For more information about the syntax, see [Multiprocessor Syntax](#). You can specify processors only in kernel mode.

Thread

Specifies the thread that the registers are read from. If you do not specify a thread, the current thread is used. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

M Mask

Specifies the mask to use when the debugger displays the registers. The "M" must be an uppercase letter. *Mask* is a sum of bits that indicate something about the register display. The meaning of the bits depends on the processor and the mode (see the tables in the following Remarks section for more information). If you omit M, the default mask is used. You can set or display the default mask by using the [Rm \(Register Mask\)](#) command.

F

Displays the floating-point registers. The "F" must be an uppercase letter. This option is equivalent to **M 0x4**.

X

Displays the SSE XMM registers. This option is equivalent to **M 0x40**.

Y

Displays the AVX YMM registers. This option is equivalent to **M 0x200**.

YI

Displays the AVX YMM integer registers. This option is equivalent to **M 0x400**.

Z

Displays the AVX-512 YMM registers (zmm0-zmm31) in floating point format.

ZI

Displays the AVX-512 YMM registers (zmm0-zmm31) in integer format.

K

Display the AVX-512 Opmask predicate registers (K0-K7).

?

(Pseudo-register assignment only) Causes the pseudo-register to acquire typed information. Any type is permitted. For more information about the **r?** syntax, see [Debugger Command Program Examples](#).

Register

Specifies the register, flag, pseudo-register, or fixed-name alias to display or modify. You must not precede this parameter with at (@) sign. For more information about the syntax, see [Register Syntax](#).

Num

Specifies the number of elements to display. If you omit this parameter but you include *Type*, the full register length is displayed.

Type

Specifies the data format to display each register element in. You can use *Type* only with 64-bit and 128-bit vector registers. You can specify multiple types.

You can specify one or more of the following values.

Type	Display format
ib	Signed byte

Type	Display format
ub	Unsigned byte
iw	Signed word
uw	Unsigned word
id	Signed DWORD
ud	Unsigned DWORD
iq	Signed quad-word
uq	Unsigned quad-word
f	32-bit floating-point
d	64-bit floating-point

Value

Specifies the value to assign to the register. For more information about the syntax, see [Numerical Expression Syntax](#).

.

Displays the registers used in the current instruction. If no registers are used, no output is displayed.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the register context and other context settings, see [Changing Contexts](#).

Remarks

If you do not specify *Register*, the **r** command displays all the non-floating-point registers, and the **rF** command displays all the floating-point registers. You can change this behavior by using the [rm \(Register Mask\)](#) command.

If you specify *Register* but you omit the equal sign (=) and the *Value* parameter, the command displays the current value of the register.

If you specify *Register* and an equal sign (=) but you omit *Value*, the command displays the current value of the register and prompts for a new value.

If you specify *Register*, the equal sign (=), and *Value*, the command changes the register to contain the value. (If *quiet mode* is active, you can omit the equal sign. You can turn on quiet mode by using the [sq \(Set Quiet Mode\)](#) command. In kernel mode, you can also turn on quiet mode by using the KDQUIET environment variable.)

You can specify multiple registers, separated by commas.

In user mode, the **r** command displays registers that are associated with the current thread. For more information about the threads, see [Controlling Processes and Threads](#).

In kernel mode, the **r** command displays registers that are associated with the current *register context*. You can set the register context to match a specific thread, context record, or trap frame. Only the most important registers for the specified register context are actually displayed, and you cannot change their values. For more information about register context, see [Register Context](#).

When you specify a floating-point register by name, the **F** option is not required. When you specify a single floating-point register, the raw hexadecimal value is displayed in addition to the decimal value.

The following *Mask* bits are supported for an x86-based processor or an x64-based processor.

Bit	Value	Description
0 1	0x1 0x2	Displays the basic integer registers. (Setting one or both of these bits has the same effect.)
2	0x4	Displays the floating-point registers.
3	0x8	Displays the segment registers.

Bit	Value	Description
4	0x10	Displays the MMX registers.
5	0x20	Displays the debug registers. In kernel mode, setting this bit also displays the CR4 register.
6	0x40	Displays the SSE XMM registers.
7	0x80	(Kernel mode only) Displays the control registers, for example CR0, CR2, CR3 and CR8.
8	0x100	(Kernel mode only) Displays the descriptor and task state registers.
9	0x200	Displays the AVX YMM registers in floating point.
10	0x400	Displays the AVX YMM registers in decimal integers.
11	0x800	Displays the AVX XMM registers in decimal integers.

The following code examples show `r` commands for an x86-based processor.

In kernel mode, the following command shows the registers for processor 2.

```
dbgcmd
1: kd> 2r
```

In user mode, the following command shows the registers for thread 2.

```
dbgcmd
0:000> ~2 r
```

In user mode, the following command displays all of the `eax` registers that are associated with all threads (in thread index order).

```
dbgcmd
```

```
0:000> ~* r eax
```

The following command sets the **eax** register for the current thread to 0x000000FF.

```
dbgcmd
```

```
0:000> r eax=0x000000FF
```

The following command sets the **st0** register to 1.234e+10 (the **F** is optional).

```
dbgcmd
```

```
0:000> rF st0=1.234e+10
```

The following command displays the zero flag.

```
dbgcmd
```

```
0:000> r zf
```

The following command displays the **xmm0** register as 16 unsigned bytes and then displays the full contents of the **xmm1** register in double-precision floating-point format.

```
dbgcmd
```

```
0:000> r xmm0:16ub, xmm1:d
```

If the current syntax is C++, you must precede registers by an at sign (@). Therefore, you could use the following command to copy the **ebx** register to the **eax** register.

```
dbgcmd
```

```
0:000> r eax = @ebx
```

The following command displays pseudo-registers in the same way that the **r** command displays registers.

```
dbgcmd
```

```
0:000> r $teb
```

You can also use the `r` command to create *fixed-name aliases*. These aliases are not registers or pseudo-registers, even though they are associated with the `r` command. For more information about these aliases, see [Using Aliases](#).

Here is an example of the `r.` command on an x86-based processor. The last entry of the call stack precedes the command itself.

```
dbgcmd  
01004af3 8bec          mov      ebp,esp  
0:000> r.  
ebp=0006ffc0  esp=0006ff7c
```

rdmsr (Read MSR)

Article • 10/25/2023

The **rdmsr** command reads a [Model-Specific Register \(MSR\)](#) value from the specified address.

dbgcmd
rdmsr Address

Parameters

Address

Specifies the address of the MSR.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Remarks

The **rdmsr** command can display MSR's on x86-based and x64-based platforms. The MSR definitions are platform-specific.

See also

[wrmsr \(Write MSR\)](#)

rm (Register Mask)

Article • 10/25/2023

The **rm** command modifies or displays the register display mask. This mask controls how registers are displayed by the [r \(Registers\)](#) command.

```
dbgcmd
```

```
rm  
rm ?  
rm Mask
```

Parameters

?

Displays a list of possible *Mask* bits.

Mask

Specifies the mask to use when the debugger displays the registers. *Mask* is a sum of bits that indicate something about the register display. The meaning of the bits depends on the processor and the mode. For more information; see the tables in the following Remarks section.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The "m" in the command name must be a lowercase letter.

If you use **rm** with no parameters, the current value is displayed, along with an explanation about its bits.

To display the basic integer registers, you must set bit 0 (0x1) or bit 1 (0x2). By default, 0x1 is set for 32-bit targets and 0x2 is set for 64-bit targets. You cannot set these two bits at the same time--if you try to set both bits, 0x2 overrides 0x1.

You can override the default mask by using the [r \(Registers\)](#) command together with the **M** option.

The following *Mask* bits are supported for an x86-based processor or an x64-based processor.

Bit	Value	Description
0 1	0x1 0x2	Displays the basic integer registers. (Setting one or both of these bits has the same effect.)
2	0x4	Displays the floating-point registers.
3	0x8	Displays the segment registers.
4	0x10	Displays the MMX registers.
5	0x20	Displays the debug registers. In kernel mode, setting this bit also displays the CR4 register.
6	0x40	Displays the SSE XMM registers.
7	0x80	(Kernel mode only) Displays the control registers, for example CR0, CR2, CR3 and CR8.
8	0x100	(Kernel mode only) Displays the descriptor and task state registers.
9	0x200	Displays the AVX YMM registers in floating point.
10	0x400	Displays the AVX YMM registers in decimal integers.
11	0x800	Displays the AVX XMM registers in decimal integers.

Bit	Value	Description
12	0x1000	Displays the AVX-512 zmm0-zmm31 registers in floating point format.
13	0x2000	Displays the AVX-512 zm00-zmm31 registers in integer format.
14	0x4000	Displays the AVX-512 k0-k7 registers.

Examples

Enable the integer state and segment registers.

```
dbgcmd
0: kd> rm 0x00a
0: kd> rm
Register output mask is a:
 2 - Integer state (64-bit)
 8 - Segment registers
```

Enable 0x1000 (Displays the AVX-512 zmm0-zmm31 registers in floating point format).

```
dbgcmd
0: kd> rm 0x100a
0: kd> rm
Register output mask is 100a:
 2 - Integer state (64-bit)
 8 - Segment registers
1000 - AVX-512 ZMM registers
```

Enable mask 0x2000 (Displays the AVX-512 zm00-zmm31 registers in integer format).

```
dbgcmd
0: kd> rm 0x200a
0: kd> rm
Register output mask is 200a:
 2 - Integer state (64-bit)
 8 - Segment registers
2000 - AVX-512 ZMM Integer registers
```

Enable all AVX-512 register masks:

```
dbgcmd

0: kd> rm 0x700a
0: kd> rm
Register output mask is 700a:
  2 - Integer state (64-bit)
  8 - Segment registers
1000 - AVX-512 ZMM registers
2000 - AVX-512 ZMM Integer registers
4000 - AVX-512 Opmask registers
```

If you try and set a register mask on hardware that does not support it, the invalid bits of the register mask will be ignored.

```
dbgcmd

kd> rm 0x100a
Ignored invalid bits 1000
kd> rm
Register output mask is a:
  2 - Integer state (64-bit)
  8 - Segment registers
```

s (Search Memory)

Article • 10/25/2023

The **s** command searches through memory to find a specific byte pattern.

Do not confuse this command with the [~s \(Change Current Processor\)](#), [~s \(Set Current Thread\)](#), [|s \(Set Current Process\)](#), or [||s \(Set Current System\)](#) commands.

dbgcmd

```
s [-[[Flags]Type]] Range Pattern
s -[[Flags]]v Range Object
s -[[Flags]]sa Range
s -[[Flags]]su Range
```

Parameters

[Flags]

Specifies one or more search options. Each flag is a single letter. You must enclose the flags in a single set of brackets ([]). You cannot add spaces between the brackets, except between **n** or **r** and its argument. For example, if you want to specify the **s** and **w** options, use the command **s -[sw]Type Range Pattern**.

You can specify one or more of the following flags:

s

Saves all of the results of the current search. You can use these results to repeat the search later.

r

Restricts the current search to the results from the last saved search. You cannot use the **s** and **r** flags in the same command. When you use **r**, the value of *Range* is ignored, and the debugger searches only those hits that were saved by the previous **s** command.

n Hits

Specifies the number of hits to save when you use the **s** flag. The default value is 1024 hits. If you use **n** together with other flags, **n** must be the last flag, followed by its *Hits* argument. The space between **n** and *Hits* is optional, but you cannot add any other spaces within the brackets. If any later search that uses the **s** flag discovers more than the specified number of hits, the **Overflow error** message is displayed to notify you that not all hits are being saved.

I *Length*

Causes a search for arbitrary ASCII or Unicode strings to return only strings that are at least *Length* characters long. The default length is 3. This value affects only searches that use the **-sa** or **-su** flags.

w

Searches only writeable memory regions. You must enclose the "w" in brackets.

1

Displays only the addresses of search matches in the search output. This option is useful if you are using the [.foreach](#) token to pipe the command output into another command's input.

Type

Specifies the memory type to search for. Add a hyphen (-) in front of *Type*. You can use one of the following *Type* values.

Type	Description
b	Byte (8 bits)
w	WORD (16 bits)
d	DWORD (32 bits)
q	QWORD (64 bits)
a	ASCII string (not necessarily a null-terminated string)
u	Unicode string (not necessarily a null-terminated string)

If you omit *Type*, byte values are used. However, if you use *Flags*, you cannot omit *Type*.

sa

Searches for any memory that contains printable ASCII strings. Use the **I** *Length* flag to specify a minimum length of such strings. The default minimum length is 3 characters.

su

Searches for any memory that contains printable Unicode strings. Use the **I** *Length* flag to specify a minimum length of such strings. The default minimum length is 3 characters.

Range

Specifies the memory area to search through. This range cannot be more than 256 MB

long unless you use the L? syntax. For more information about this syntax, see [Address and Address Range Syntax](#).

Pattern

Specifies one or more values to search for. By default, these values are byte values. You can specify different types of memory in *Type*. If you specify a WORD, DWORD, or QWORD value, depending on the other options selected, you may need to enclose the search pattern in single quotation marks (for example, 'H').

```
dbgcmd
```

```
0:000> s -d @rsp L1000000 'H'
0000003f`ff07ec00 00000048 00000000 500c0163 00000000 H.....c..P....
0000003f`ff07ec50 00000048 00000000 00000080 00000000 H.....
0000003f`ff07efc0 00000048 00000000 400c0060 00000000 H.....`..@....
```

If you specify a string, using the ascii type, enclose it in double quotation marks (for example, "B7").

```
dbgcmd
```

```
0:000> s -a @rsp L10000000 "B7"
0000003f`ff07ef0a 42 37 ff 7f 00 00 90 38-4e c2 6c 01 00 00 7d 26
B7.....8N.1...}&
0000003f`ff0ff322 42 37 ff 7f 00 00 f8 5d-42 37 ff 7f 00 00 20 41
B7.....]B7.... A
0000003f`ff0ff32a 42 37 ff 7f 00 00 20 41-42 37 ff 7f 00 00 98 59 B7....
AB7.....Y
```

-v

Searches for objects of the same type as the specified *Object*.

Object

Specifies the address of an object or the address of a pointer to an object. The debugger then searches for objects of the same type as the object that *Object* specifies.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

If the debugger finds the byte pattern that you specify, the debugger displays the first memory address in the *Range* memory area where the pattern was found. The debugger displays an excerpt of memory that begins at that location in a format that matches the specified *Type* memory type. If *Type* is **a** or **u**, the memory contents and the corresponding ASCII or Unicode characters are displayed.

You must specify the *Pattern* parameter as a series of bytes, unless you specify a different *Type* value. You can enter byte values as numeric or ASCII characters:

- Numeric values are interpreted as numbers in the current radix (16, 10, or 8). To change the default radix, use the [n \(Set Number Base\)](#) command. You can override the default radix by specifying the **0x** prefix (hexadecimal), the **0n** prefix (decimal), the **0t** prefix (octal), or the **0y** prefix (binary). **Note** The default radix behaves differently when you use C++ expressions. For more information about these expressions and the radix, see [Evaluating Expressions](#).
- You must enclose ASCII characters in single straight quotation marks. You cannot use C-style escape characters (such as '`\0`' or '`\n`').

If you specify multiple bytes, you must separate them by spaces.

The **s-a** and **s-u** commands search for specified ASCII and Unicode strings, respectively. These strings do not have to be null-terminated.

The **s-sa** and **s-su** commands search for unspecified ASCII and Unicode strings. These are useful if you are checking a range of memory to see whether it contains any printable characters. The flags options allow you to specify a minimum length of string to find.

Example: The following command finds ASCII strings that are of length ≥ 3 in the range beginning at `0000000140000000` and ending 400 bytes later.

```
dbgcmd
```

```
s-sa 0000000140000000 L400
```

The following command finds ASCII strings that are of length >=4 in the range beginning at 0000000140000000 and ending 400 bytes later

```
dbgcmd  
s -[14]sa 0000000140000000 L400
```

The following command does the same thing, but it limits the search to writeable memory regions.

```
dbgcmd  
s -[w14]sa 0000000140000000 L400
```

The following command does the same thing, but displays only the address of the match, rather than the address and the value.

```
dbgcmd  
s -[1w14]sa 0000000140000000 L400
```

The **s-v** command searches for objects of the same data type as the *Object* object. You can use this command only if the desired object is a C++ class or another object that is associated with virtual function tables (Vtables). The **s-v** command searches the *Range* memory area for the addresses of this class's Vtables. If multiple Vtables exist in this class, the search algorithm looks for all of these pointer values, separated by the proper number of bytes. If any matches are found, the debugger returns the base address of the object and full information about this object--similar to the output of the [dt \(Display Type\)](#) command.

Example: Assume the current radix is 16. The following three command all do the same thing: search memory locations 0012FF40 through 0012FF5F for "Hello".

```
dbgcmd  
0:000> s 0012ff40 L20 'H' 'e' 'l' 'l' 'o'
```

```
dbgcmd  
0:000> s 0012ff40 L20 48 65 6c 6c 6f
```

```
dbgcmd
```

```
0:000> s -a 0012ff40 L20 "Hello"
```

These commands locate each appearance of "Hello" and return the address of each such pattern--that is, the address of the letter "H".

The debugger returns only patterns that are completely contained in the search range. Overlapping patterns are found correctly. (In other words, the pattern "QQQ" is found three times in "QQQQQ".)

The following example shows a search that uses the *Type* parameter. This command searches memory locations 0012FF40 through 0012FF5F for the double-word 'VUTS':

```
dbgcmd
```

```
0:000> s -d 0012ff40 L20 'VUTS'
```

On little-endian computers, 'VUTS' is the same as the byte pattern 'S' 'T' 'U' 'V'. However, searches for WORDs, DWORDs, and QWORDs return only results that are correctly byte-aligned.

so (Set Kernel Debugging Options)

Article • 04/03/2024

The **so** command sets or displays the kernel debugging options.

```
dbgcmd
```

```
  so [Options]
```

Parameters

Options One or more of the following options:

NOEXTWARNING

Does not issue a warning when the debugger cannot find an extension command.

NOVERSIONCHECK

Does not check the version of debugger extension DLLs.

If you omit *Options*, the current options are displayed.

Environment

[+] Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Remarks

You can also set kernel debugging options using the [_NT_DEBUG_OPTIONS environment variable](#).

Feedback

Was this page helpful?

 Yes

 No

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

sq (Set Quiet Mode)

Article • 10/25/2023

The **sq** command turns quiet mode on or off.

```
dbgcmd
```

```
sq  
sq{e|d}
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **sqe** command turns quiet mode on, and the **sqd** command turns it off. The **sq** command turns on and off quiet mode. Each of these commands also displays the new quiet mode status.

You can set quiet mode in KD or kernel-mode WinDbg by using the [KDQUIET environment variable](#). (Note that quiet mode exists in both user-mode and kernel-mode debugging, but the KDQUIET environment variable is only recognized in kernel mode.)

Quiet mode has three distinct effects:

- The debugger does not display messages every time that an extension DLL is loaded or unloaded.
- The [r \(Registers\)](#) command no longer requires an equal sign (=) in its syntax.
- When you break into a target computer while kernel debugging, the long warning message is suppressed.

Do not confuse quiet mode with the effects of the [-myob command-line option](#) (in CDB and KD) or the [-Q command-line option](#) (in WinDbg).

ss (Set Symbol Suffix)

Article • 10/25/2023

The **ss** command sets or displays the current suffix value that is used for symbol matching in numeric expressions.

```
dbgcmd
```

```
ss [a|w|n]
```

Parameters

a

Specifies that the symbol suffix should be "A", matching many ASCII symbols.

w

Specifies that the symbol suffix should be "W", matching many Unicode symbols.

n

Specifies that the debugger should not use a symbol suffix. (This parameter is the default behavior.)

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about symbol matching, see [Symbol Syntax and Symbol Matching](#).

Remarks

If you specify the **ss** command together with no parameters, the current state of the suffix value is displayed.

sx, sxd, sxe, sxi, sxn, sxr, sx- (Set Exceptions)

Article • 10/25/2023

The **sx** commands control the action that the debugger takes when an exception occurs in the application that is being debugged, or when certain events occur.

```
dbgcmd

sx

sx{e|d|i|n} [-c "Cmd1"] [-c2 "Cmd2"] [-h] {Exception|Event|*}

sx- [-c "Cmd1"] [-c2 "Cmd2"] {Exception|Event|*}

sxr
```

Parameters

-c "Cmd1"

Specifies a command that's executed if the exception or event occurs. This command is executed when the first chance to handle this exception occurs, regardless of whether this exception breaks into the debugger. You must enclose the *Cmd1* string in quotation marks. This string can include multiple commands if you separate them with semicolons. The space between the -c and the quoted command string is optional.

-c2"Cmd2"

Specifies a command that's executed if the exception or event occurs and isn't handled on the first chance. This command is executed when the second chance to handle this exception occurs, regardless of whether this exception breaks into the debugger. You must enclose the *Cmd2* string in quotation marks. This string can include multiple commands if you separate them with semicolons. The space between the -c2 and the quoted command string is optional.

-h

Changes the specified event's handling status instead of its break status. If *Event* is **cc**, **hc**, **bpec**, or **ssec**, you don't have to use the -h option.

Exception

Specifies the exception number that the command acts on in the current radix.

Event

Specifies the event that the command acts on. These events are identified by short abbreviations. For a list of the events, see [Controlling exceptions and events](#).

*

Affects all exceptions that aren't otherwise explicitly named for **sx**. For a list of explicitly named exceptions, see [Controlling exceptions and events](#).

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	All

Additional Information

For more information about break status and handling status, descriptions of all event codes, a list of the default status for all events, and other methods of controlling this status, see [Controlling exceptions and events](#).

Remarks

The **sx** command displays the list of exceptions for the current process and the list of all non-exception events and displays the default behavior of the debugger for each exception and event.

The **sxe**, **sxd**, **sxn**, and **sxi** commands control the debugger settings for each exception and event.

The **sxr** command resets all of the exception and event filter states to the default settings. For example, commands are cleared and break, and continue options are reset to their default settings.

The **sx-** command doesn't change the handling status or the break status of the specified exception or event. This command can be used if you wish to change the first-chance command or second-chance command associated with a specific event, but don't wish to change anything else.

If you include the **-h** option (or if the **cc**, **hc**, **bpec**, or **ssec** events are specified), the **sxe**, **sxd**, **sxn**, and **sxi** commands control the [handling status](#) of the exception or event. In all other cases, these commands control the [break status](#) of the exception or event.

When you're setting the break status, these commands have the following effects:

Command	Status name	Description
sxe	Break (Enabled)	When this exception occurs, the target immediately breaks into the debugger before any other error handlers are activated. This kind of handling is called <i>first-chance</i> handling.
sxd	Second chance break (Disabled)	The debugger doesn't break for a first-chance exception of this type (although a message is displayed). If other error handlers don't address this exception, the execution stops and the target breaks into the debugger. This kind of handling is called <i>second-chance</i> handling.
sxn	Output (Notify)	When this exception occurs, the target application doesn't break into the debugger at all. However, a message is displayed that notifies the user of this exception.
sxi	Ignore	When this exception occurs, the target application doesn't break into the debugger at all, and no message is displayed.

When you're setting the handling status, these commands have the following effects:

Command	Status name	Description
sxe	Handled	The event is considered handled when execution resumes.
sxd,sxn,sxi	Not Handled	The event is considered not handled when execution resumes.

You can use the **-h** option together with exceptions, not events. Use this option with **ch**, **bpe**, or **sse** to set the handling status for **hc**, **bpec**, or **ssec**, respectively. If you use the **-h** option with any other event, it has no effect.

Use the **-c** or **-c2** options with **hc**, **bpec**, or **ssec** to associate the specified commands with **ch**, **bpe**, or **sse**, respectively.

In the following example, the **sxe** command is used to set the break status of access violation events to break on the first chance, and to set the first-chance command that will be executed at that point to **r eax**. Then the **sx-** command is used to alter the first-chance command to **r ebx**, without changing the handling status. Finally, a portion of the **sx** output is shown, indicating the current settings for access violation events:

```
dbgcmd
```

```
0:000> sxe -c "r eax" av  
  
0:000> sx- -c "r ebx" av  
  
0:000> sx  
av - Access violation - break - not handled  
Command: "r ebx"  
. . .
```

See also

- [Using breakpoints - debugging techniques](#)
- [Conditional breakpoints in WinDbg](#)
- [Executing until a specified state is reached](#)

t (Trace)

Article • 10/25/2023

The **t** command executes a single instruction or source line and optionally displays the resulting values of all registers and flags. When subroutine calls or interrupts occur, each of their steps is also traced.

User-Mode

```
dbgcmd  
[~Thread] t [r] [= StartAddress] [Count] ["Command"]
```

Kernel-Mode

```
dbgcmd  
t [r] [= StartAddress] [Count] ["Command"]
```

Parameters

Thread

Specifies threads to thaw. All other threads are frozen. For more information about this syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the [pr](#), [tr](#), or [.prompt_allow -reg](#) commands. All three of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the [l-os](#) command. This setting is separate from the other three commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where execution should begin. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of instructions or source lines to trace through before stopping. Each step is displayed as a separate action in the [Debugger Command window](#). The default value is one.

Command

Specifies a debugger command to execute after the trace is performed. This command is executed before the standard **t** results are displayed. If you also use *Count*, this command is executed after all tracing is complete (but before the results from the final trace are displayed).

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about how to issue the **t** command and an overview of related commands, see [Controlling the Target](#).

Remarks

When you specify *Count*, each instruction is displayed as it is stepped through.

Each trace executes a single assembly instruction or a single source line, depending on whether the debugger is in assembly mode or source mode. Use the **I+t** and **I-t** commands or the buttons on the WinDbg toolbar to switch between these modes.

If you want to trace most function calls but skip certain calls, you can use [.step_filter \(Set Step Filter\)](#) to indicate which calls to step over.

You can use the **t** command to trace instructions in ROM.

When you are quickly tracing many times in WinDbg, the debugging information windows are updated after each trace. If this update causes slower response time, use [.suspend_ui \(Suspend WinDbg Interface\)](#) to temporarily suspend the updating of these windows.

ta (Trace to Address)

Article • 10/25/2023

The **ta** command executes the program until the specified address is reached, displaying each step (including steps within called functions).

User-Mode

```
dbgcmd  
[~Thread] ta [r] [= StartAddress] StopAddress
```

Kernel-Mode

```
dbgcmd  
ta [r] [= StartAddress] StopAddress
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tar**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and use of any of them overrides any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the [**rm \(Register Mask\)**](#) command.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

StopAddress

Specifies the address at which execution stops. This address must match the exact address of an instruction.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **ta** command causes the target to begin executing. This execution continues until the specified instruction is reached or a breakpoint is encountered.

Note If you use the **ta** command in kernel mode, execution stops when an instruction is encountered at the specified virtual address in any virtual address space.

During this execution, all steps are displayed explicitly. If a function is called, the debugger also traces through that function. Therefore, the display of this command resembles what you see if you executed **t (Trace)** repeatedly until the program counter reached the specified address.

For example, the following command explicitly traces through the target code until the return address of the current function is reached.

```
dbgcmd
0:000> ta @$ra
```

tb (Trace to Next Branch)

Article • 10/25/2023

The **tb** command executes the program until a branch instruction is reached.

dbgcmd

tb [r] [= StartAddress] [Count]

Parameters

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tbr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-*os*** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the **rm (Register Mask)** command.

StartAddress

Specifies the address where the debugger starts execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of branches to allow. Every time that a branch is encountered, the instruction address and the instruction are displayed. If you omit *Count*, the default number is 1.

Environment

Modes	x86-based: Kernel mode only x64-based: User mode, kernel mode
Targets	Live debugging only
Platforms	x86-based (GenuineIntel processor family 6 and later), x64-based

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **tb** command causes the target to begin executing. This execution continues until a branch command is reached.

Execution stops at any branch command that is to be taken. This stopping of execution is always based on the *disassembly* code, even when the debugger is in source mode.

Branch instructions include calls, returns, jumps, counted loops, and while loops. If the debugger encounters an unconditional branch, or a conditional branch for which the condition is true, execution stops. If the debugger encounters a conditional branch whose condition is false, execution continues.

When execution stops, the address of the branch instruction and any associated symbols are displayed. This information is followed by an arrow and then the address and instructions of the new program counter location.

The **tb** command works only on the current processor. If you use **tb** on a multiprocessor system, execution stops when a branch command is reached or when another processor's event occurs, whichever comes first.

Usually, branch tracing is enabled after the processor control block (PRCB) has been initialized. (The PRCB is initialized early in the boot process.) However, if you have to use the **tb** command before this point, you can use [.force_tb \(Forcibly Allow Branch Tracing\)](#) to enable branch tracing earlier. Use the **.force_tb** command cautiously, because it can corrupt your processor state.

tc (Trace to Next Call)

Article • 10/25/2023

The **tc** command executes the program until a call instruction is reached.

User-Mode

```
dbgcmd  
[~Thread] tc [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
tc [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tcr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **call** instructions that the debugger must encounter for the **tc**

command to end. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **tc** command causes the target to begin executing. This execution continues until the debugger reaches a **call** instruction or encounters a breakpoint.

If the program counter is already on a **call** instruction, the debugger traces into the call and continues executing until it encounters another **call**. This tracing, rather than execution, of the call is the only difference between **tc** and [pc \(Step to Next Call\)](#).

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **call** instruction that is associated with the current source line.

tct (Trace to Next Call or Return)

Article • 10/25/2023

The **tct** command executes the program until it reaches a call instruction or return instruction.

User-Mode

```
dbgcmd  
[~Thread] tct [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
tct [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **tctr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the [**rm \(Register Mask\)**](#) command.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **call** or **return** instructions that the debugger must encounter for the **tct** command to end. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **tct** command causes the target to begin executing. This execution continues until the debugger reaches a **call** or **return** instruction or encounters a breakpoint.

If the program counter is already on a **call** or **return** instruction, the debugger traces into the call or return and continues executing until it encounters another **call** or **return**. This tracing, rather than execution, of the call is the only difference between **tct** and [**pct** \(Step to Next Call or Return\)](#).

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **call** or **return** instruction that is associated with the current source line.

th (Trace to Next Branching Instruction)

Article • 10/25/2023

The **th** command executes the program until it reaches any kind of branching instruction, including conditional or unconditional branches, calls, returns, and system calls.

User-Mode

```
dbgcmd  
[~Thread] th [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
th [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **thr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of branching instructions that the debugger must encounter for the **th** command to end. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **th** command causes the target to begin executing. Execution continues until the debugger reaches a branching instruction or encounters a breakpoint.

If the program counter is already on a branching instruction, the debugger traces into the branching instruction and continues executing until another branching instruction is reached. This tracing, rather than execution, of the call is the only difference between **th** and [ph \(Step to Next Branching Instruction\)](#).

th is available for all live sessions. This availability is the primary difference between **th** and [tb \(Trace to Next Branch\)](#).

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a branching instruction that is associated with the current source line.

tt (Trace to Next Return)

Article • 10/25/2023

The **tt** command executes the program until a return instruction is reached.

User-Mode

```
dbgcmd  
[~Thread] tt [r] [= StartAddress] [Count]
```

Kernel-Mode

```
dbgcmd  
tt [r] [= StartAddress] [Count]
```

Parameters

Thread

Specifies threads to continue executing. All other threads are frozen. For more information about the syntax, see [Thread Syntax](#). You can specify threads only in user mode.

r

Turns on and off the display of registers and flags. By default, the registers and flags are displayed. You can disable register display by using the **ttr**, **pr**, **tr**, or **.prompt_allow -reg** commands. All of these commands control the same setting and you can use any of them to override any previous use of these commands.

You can also disable register display by using the **l-os** command. This setting is separate from the other four commands. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

Count

Specifies the number of **return** instructions that the debugger must encounter for the **th**

command to end. The default value is one.

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For more information about related commands, see [Controlling the Target](#).

Remarks

The **tt** command causes the target to begin executing. This execution continues until the debugger reaches a **return** instruction or encounters a breakpoint.

If the program counter is already on a **return** instruction, the debugger traces into the return and continues executing until another **return** is reached. This tracing, rather than execution, of the call is the only difference between **tt** and [pt \(Step to Next Return\)](#).

In source mode, you can associate one source line with multiple assembly instructions. This command does not stop at a **return** instruction that is associated with the current source line.

u, ub, uu (Unassemble)

Article • 10/25/2023

The **u*** commands display an assembly translation of the specified program code in memory.

This command should not be confused with the [~u \(Unfreeze Thread\)](#) command.

dbgcmd

```
u[u|b] Range  
u[u|b] Address  
u[u|b]
```

Parameters

Range

Specifies the memory range that contains the instructions to disassemble. For more information about the syntax, see [Address and Address Range Syntax](#). If you use the **b** flag, you must specify *Range* by using the "Address LLength" syntax, not the "Address1 Address2" syntax.

Address

Specifies the beginning of the memory range to disassemble. Eight instructions on an x86-based processor are unassembled. For more information about the syntax, see [Address and Address Range Syntax](#).

b

Determines the memory range to disassemble by counting backward. If **ub** *Address* is used, the disassembled range will be the eight or nine byte range ending with *Address*. If a range is specified using the syntax **ub** *Address* *LLength*, the disassembled range will be the range of the specified length ending at *Address*.

u

Specifies that the disassembly will continue even if there is a memory read error.

Environment

Item	Description
Modes	User mode, kernel mode

Item	Description
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

If you do not specify a parameter for the **u** command, the disassembly begins at the current address and extends eight instructions on an x86-based or x64-based processor. When you use **ub** without a parameter, the disassembly includes the eight or nine instructions before the current address.

Do not confuse this command with the [up \(Unassemble from Physical Memory\)](#). The **u** command disassembles only virtual memory, while the **up** command disassembles only physical memory.

uf (Unassemble Function)

Article • 10/25/2023

The **uf** command displays an assembly translation of the specified function in memory.

```
dbgcmd
```

```
uf [Options] Address
```

Parameters

Options One or more of the following options:

/c

Displays only the call instructions in a routine instead of the full disassembly. Call instructions can be useful for determination of caller and callee relationships from disassembled code.

/D

Creates linked callee names for navigation of the call graph.

/m

Relaxes the blocking requirements to permit multiple exits.

/o

Sorts the display by address instead of by function offset. This option presents a memory-layout view of a full function.

/O

Creates linked call lines for accessing call information and creating breakpoints.

/i

Displays the number of instructions in a routine.

Address

Specifies the address of the function to disassemble. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

The display shows the whole function, according to the function order.

up (Unassemble from Physical Memory)

Article • 10/25/2023

The **up** command displays an assembly translation of the specified program code in physical memory.

```
dbgcmd  
  up Range  
  up Address  
  up
```

Parameters

Range

Specifies the memory range in physical memory that contains the instructions to disassemble. For more information about the syntax, see [Address and Address Range Syntax](#).

Address

Specifies the beginning of the memory range in physical memory to disassemble. Eight instructions on an x86-based processor are unassembled. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

If you do not specify a parameter for the **up** command, the disassembly begins at the current address and extends eight instructions on an x86-based processor.

Do not confuse this command with the [u \(Unassemble\)](#). The **up** command disassembles only physical memory, while the **u** command disassembles only virtual memory.

ur (Unassemble Real Mode BIOS)

Article • 10/25/2023

The **ur** command displays an assembly translation of the specified 16-bit real-mode code.

dbgcmd

```
ur Range  
ur Address  
ur
```

Parameters

Range

Specifies the memory range that contains the instructions to disassemble. For more information about the syntax, see [Address and Address Range Syntax](#).

Address

Specifies the beginning of the memory range to disassemble. Eight instructions on an x86-based processor are unassembled. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to debug BIOS code, see [Debugging BIOS Code](#).

Remarks

If you do not specify *Range* or *Address*, the disassembly begins at the current address and extends eight instructions on an x86-based processor.

If you are examining 16-bit real-mode code on an x86-based processor, both the **ur** command and the [u \(Unassemble\)](#) command give correct results.

However, if real-mode code exists in a location where the debugger is not expecting it (for example, a non-x86 computer that is running or emulating x86-based BIOS code from a plug-in card), you must use **ur** to correctly disassemble this code.

If you use **ur** on 32-bit or 64-bit code, the command tries to disassemble the code as if it were 16-bit code. This situation produces meaningless results.

ux (Unassemble x86 BIOS)

Article • 10/25/2023

The **ux** command displays the instruction set of the x86-based BIOS code.

```
ux [Address]
```

Parameters

Address

Specifies the memory offset within the x86-based BIOS code. If you omit this parameter or specify zero, the default offset is the beginning of the BIOS.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based only

Additional Information

For more information about how to debug BIOS code, see [Debugging BIOS Code](#).

Remarks

The debugger displays the instructions that are generated from the first eight lines of code, beginning at the *Address* offset.

To make the **ux** command work correctly, HAL symbols must be available to the debugger. If the debugger cannot find these symbols, the debugger displays a "couldn't resolve" error.

vercommand (Show Debugger Command Line)

Article • 10/25/2023

The **vercommand** command displays the command that opened the debugger.

`vercommand`

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

version (Show Debugger Version)

Article • 10/25/2023

The **version** command displays version information about the debugger and all loaded extension DLLs. This command also displays the current version of the operating system of the target computer.

Do not confuse this command with the [!version \(Show DLL Version\)](#) extension command.

`version`

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

[CTRL+W \(Show Debugger Version\)](#)

[vertarget \(Show Target Computer Version\)](#)

vertarget (Show Target Computer Version)

Article • 10/25/2023

The **vertarget** command displays the current version of the Microsoft Windows operating system of the target computer.

`vertarget`

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

See also

[CTRL+W \(Show Debugger Version\)](#)

[version \(Show Debugger Version\)](#)

wrmsr (Write MSR)

Article • 04/02/2024

The **wrmsr** command writes a value to a Model-Specific Register (MSR) at the specified address.

`wrmsr Address Value`

Parameters

Address

Specifies the address of the MSR.

Value

Specifies the 64-bit hexadecimal value to write to the MSR.

Environment

[+] Expand table

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Remarks

The **wrmsr** command can set MSR's on x86-based and x64-based platforms. The MSR definitions are platform-specific.

See also

[rdmsr \(Read MSR\)](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

wt (Trace and Watch Data)

Article • 10/25/2023

The **wt** command runs through the whole function and then displays statistics, when you execute this command at the beginning of a function call.

dbgcmd

```
wt [WatchOptions] [= StartAddress] [EndAddress]
```

Parameters

WatchOptions

Specifies how to modify the display. You can use any of the following options.

Option	Effect
-l Depth	(User mode only) Specifies the maximum depth of the calls to display. Any calls that are at least <i>Depth</i> levels deeper than the starting point are executed silently.
-m Module	(User mode only) Restricts the display to code inside the specified module, plus the first level of calls made from that module. You can include multiple -m options to display code from multiple modules and no other modules.
-i Module	(User mode only) Ignores any code within the specified module. You can include multiple -i options to ignore code from multiple modules. If you use a -m option, the debugger ignores all -i options.
-ni	(User mode only) Does not display any entry into code that is being ignored because of an -m or -i option.
-nc	Does not display individual call information.
-ns	Does not display summary information.
-nw	Does not display warnings during the trace.
-oa	(User mode only) Displays the actual address of call sites.

Option	Effect
-or	(User mode only) Displays the return register values of the called function, using the default radix as the base.
-oR	(User mode only) Displays the return register values of the called function, in the appropriate type for each return value.

StartAddress

Specifies the address where the debugger begins execution. If you do not use *StartAddress*, execution begins at the instruction that the instruction pointer points to. For more information about the syntax, see [Address and Address Range Syntax](#).

EndAddress

Specifies the address where tracing ends. If you do not use *EndAddress*, a single instruction or function call is executed.

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	User mode: all Kernel mode: x86-based only

Additional Information

For more information about issuing the **wt** command and an overview of related commands, see [Controlling the Target](#).

Remarks

The **wt** command is useful if you want information about the behavior of a specific function, but you do not want to step through the function. Instead, go to the beginning of that function and then issue the **wt** command.

If the program counter is at a point that corresponds to a symbol (such as the beginning of a function or entry point into a module), the **wt** command traces until it reaches the current return address. If the program counter is on a **call** instruction, the **wt** command traces until it returns to the current location. This tracing is profiled in the [Debugger](#)

Command window together with output that describes the various calls that the command encounters.

If the **wt** command is issued somewhere other than the beginning of a function, the command behaves like the **p (Step)** command. However, if you specify *EndAddress*, execution continues until that address is reached, even if this execution involves many program steps and function calls.

When you are debugging in source mode, you should trace into the function only to the point where you see the opening bracket of the function body. Then, you can use the **wt** command. (It is typically easier to insert a breakpoint at the first line of the function, or use **Debug | Run to Cursor**, and then use the **wt** command.)

Because the output from **wt** can be long, you might want to use a log file to record your output.

The following example shows a typical log file.

```
dbgcmd

0:000> l+                               Source options set to show source lines
Source options are f:
  1/t - Step/trace by source line
  2/l - List source line for LN and prompt
  4/s - List source code at prompt
  8/o - Only show source code at prompt
0:000> p                               Not yet at the function call: use "p"
> 44:      minorVariableOne = 12;
0:000> p
> 45:      variableOne = myFunction(2, minorVariable);
0:000> t                               At the function call: now use "t"
MyModule!ILT+10(_myFunction):
0040100f e9cce60000      jmp      MyModule!myFunction (0040f6e0)
0:000> t
> 231:  {
0:000> wt                               At the function beginning: now use "wt"
Tracing MyModule!myFunction to return address 00401137

105    0 [ 0] MyModule!myFunction
  1    0 [ 1]  MyModule!ILT+1555(_printf)
  9    0 [ 1]  MyModule!printf
  1    0 [ 2]  MyModule!ILT+370(__stbuf)
 11    0 [ 2]  MyModule!_stbuf
  1    0 [ 3]  MyModule!ILT+1440(__isatty)
 14    0 [ 3]  MyModule!_isatty
 50    15 [ 2]  MyModule!_stbuf
 17    66 [ 1]  MyModule!printf
  1    0 [ 2]  MyModule!ILT+980(__output)
 59    0 [ 2]  MyModule!_output
 39    0 [ 3]  MyModule!write_char
```

```

111    39 [ 2]      MyModule!_output
 39     0 [ 3]      MyModule!write_char

....  

11     0 [ 5]      kernel32!__SEH_epilog4
54 11675 [ 4]      kernel32!ReadFile
165 11729 [ 3]      MyModule!_read
100 11895 [ 2]      MyModule!_filbuf
 91 11996 [ 1]      MyModule!fgets
54545 83789 [ 0] MyModule!myFunction
  1     0 [ 1]  MyModule!ILT+1265(__RTC_CheckEsp)
  2     0 [ 1]  MyModule!_RTC_CheckEsp
54547 83782 [ 0] MyModule!myFunction

```

112379 instructions were executed in 112378 events (0 from other threads)

Function Name	Invocations	MinInst	MaxInst
AvgInst			
MyModule!ILT+1265(__RTC_CheckEsp)	1	1	1
1			
MyModule!ILT+1440(__isatty)	21	1	1
1			
MyModule!ILT+1540(__ftbuf)	21	1	1
1			
....			
ntdll!memcpy	24	1	40
19			
ntdll!memset	2	29	29
29			

23 system calls were executed

Calls	System Call
23	ntdll!KiFastSystemCall

In the listing of the trace, the first number specifies the number of instructions that were executed, the second number specifies the number of instructions executed by child processes of the function, and the third number (in brackets) specifies the depth of the function in the stack (taking the initial function as zero). The indentation of the function name shows the call depth.

In the preceding example, **MyModule!myFunction** executes 105 instructions before it calls several subroutines, including **printf** and **fgets**, and then executes 54545 additional instructions after calling those functions, but before issuing a few more calls. However, in the final count, the display shows that **myFunction** executes 112,379 instructions, because this count includes all of the instructions that **myFunction** and its children execute. (The *children* of **myFunction** are functions that are called from **myFunction**, either directly or indirectly.)

In the preceding example, note also that **ILT+1440 (_isatty)** is called 21 times. In the final count, the summary of this function's behavior shows the number of times that it was called, the smallest number of instructions in any single execution, the largest number of instructions in any single execution, and the average number of instructions per execution.

If any system calls are made, they appear in the counter and are listed again at the end of the command output.

x (Examine Symbols)

Article • 10/25/2023

The **x** command displays the symbols in all contexts that match the specified pattern.

```
dbgcmd  
x [Options] Module!Symbol  
x [Options] *
```

Parameters

Options Specifies symbol searching options. You can use one or more of the following options:

/0

Displays only the address of each symbol.

/1

Displays only the name of each symbol.

/2

Displays only the address and name of each symbol (not the data type).

/D

Displays the output using [Debugger Markup Language](#).

/t

Displays the data type of each symbol, if the data type is known.

/v

Displays the symbol type (local, global, parameter, function, or unknown) of each symbol. This option also displays the size of each symbol. The size of a function symbol is the size of the function in memory. The size of other symbols is the size of the data type that the symbol represents. Size is always measured in bytes and displayed in hexadecimal format.

/s Size

Display only those symbols whose size, in bytes, equals the value of *Size*. The *Size* of a function symbol is the size of the function in memory. The *Size* of other symbols is the size of the data type that the symbol represents. Symbols whose size cannot be determined are always displayed. *Size* must be a nonzero integer.

/q

Displays symbol names in quoted format.

/p

Omits the space before the opening parenthesis when the debugger displays a function name and its arguments. This kind of display can make it easier if you are copying function names and arguments from the **x** display to another location.

/f

Displays the data size of a function.

/d

Displays the data size of data.

/a

Sorts the display by address, in ascending order.

/A

Sorts the display by address, in descending order.

/n

Sorts the display by name, in ascending order.

/N

Sorts the display by name, in descending order.

/z

Sorts the display by size, in ascending order.

/Z

Sorts the display by size, in descending order.

Module

Specifies the module to search. This module can be an .exe, .dll, or .sys file. *Module* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#).

Symbol

Specifies a pattern that the symbol must contain. *Symbol* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#).

Because this pattern is matched to a symbol, the match is not case sensitive, and a single leading underscore (_) represents any quantity of leading underscores. You can

add spaces within *Symbol*, so that you can specify symbol names that contain spaces (such as "operator new" or "Template<A, B>") without using wildcard characters.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The following command finds all of the symbols in MyModule that contain the string "spin".

```
dbgcmd  
0:000> x mymodule!*spin*
```

The following command quickly locates the "DownloadMinor" and "DownloadMajor" symbols in MyModule.

```
dbgcmd  
0:000> x mymodule!downloadm?or
```

You can also show all symbols in the MyModule by using the following command.

```
dbgcmd  
0:000> x mymodule!*
```

The preceding commands also force the debugger to reload symbol information from MyModule. If you want to reload the symbols in the module with a minimal display, use the following command.

```
dbgcmd  
0:000> x mymodule!*start*
```

A few symbols always contain the string "start". Therefore, the preceding command always displays some output to verify that the command works. But the preceding command avoids the excessive display length of **x mymodule!***.

The display shows the starting address of each symbol and the full symbol name. If the symbol is a function name, the display also includes a list of its argument types. If the symbol is a global variable, its current value is displayed.

There is one other special case of the **x** command. To display the addresses and names of all local variables for the current context, use the following command.

```
dbgcmd  
0:000> x *
```

Note In most cases, you cannot access local variables unless private symbols have been loaded. For more information about this situation, see [dbgerr005: Private Symbols Required](#). To display the values of local variables, use the [dv \(Display Local Variables\)](#) command.

The following example illustrates the **/0**, **/1**, and **/2** options.

```
dbgcmd  
0:000:x86> x /0 MyApp!Add*  
00b51410  
00b513d0  
  
0:000:x86> x /1 MyApp!Add*  
MyApp!AddThreeIntegers  
MyApp!AddTwoIntegers  
  
0:000:x86> x /2 MyApp!Add*  
00b51410      MyApp!AddThreeIntegers  
00b513d0      MyApp!AddTwoIntegers
```

The **/0**, **/1**, and **/2** options are useful if you want to use the output of the **x** command as input to the [.foreach](#) command.

```
dbgcmd  
.foreach ( place { x /0 MyApp!*MySym* } ) { .echo ${place}+0x18 }
```

The following example demonstrates the switch **/f** when used to filter functions on the module notepad.exe.

dbgcmd

```
0:000> x /f /v notepad!*main*
prv func 00000001`00003340 249 notepad!WinMain (struct HINSTANCE__ *,
struct HINSTANCE__ *, char *, int)
prv func 00000001`0000a7b0 1c notepad!WinMainCRTStartup$filt$0 (void)
prv func 00000001`0000a540 268 notepad!WinMainCRTStartup (void)
```

When you use the */v* option, the first column of the display shows the symbol type (local, global, parameter, function, or unknown). The second column is the address of the symbol. The third column is the size of the symbol, in bytes. The fourth column shows the module name and symbol name. In some cases, this display is followed by an equal sign (=) and then the data type of the symbol. The source of the symbol (public or full symbol information) is also displayed.

dbgcmd

```
kd> x /v nt!CmType*
global 806c9e68 0 nt!CmTypeName = struct _UNICODE_STRING []
global 806c9e68 150 nt!CmTypeName = struct _UNICODE_STRING [42]
global 806c9e68 0 nt!CmTypeName = struct _UNICODE_STRING []
global 805bd7b0 0 nt!CmTypeString = unsigned short *[]
global 805bd7b0 a8 nt!CmTypeString = unsigned short *[42]
```

In the preceding example, the size is given in hexadecimal format, while the data type is given in decimal format. Therefore, in the last line of the preceding example, the data type is an array of 42 pointers to unsigned short integers. The size of this array is $42 \times 4 = 168$, and 168 is displayed in hexadecimal format as 0xA8.

You can use the */sSize* option to display only those symbols whose size, in bytes, is a certain value. For example, you can restrict the command in the preceding example to symbols that represent objects whose size is 0xA8.

dbgcmd

```
kd> x /v /s a8 nt!CmType*
global 805bd7b0 a8 nt!CmTypeString = unsigned short *[42]
```

Working With Data Types

The */t* option causes the debugger to display information about each symbol's data type. Note that for many symbols, this information is displayed even without the */t* option. When you use */t*, such symbols have their data type information displayed twice.

dbgcmd

```
0:001> x prymes!__n*
00427d84 myModule!__nullstring = 0x00425de8 "(null)"
0042a3c0 myModule!_nstream = 512
Type information missing error for _nh_malloc
004021c1 myModule!MyStructInstance = struct MyStruct
00427d14 myModule!_NLG_Destination = <no type information>

0:001> x /t prymes!__n*
00427d84 char * myModule!__nullstring = 0x00425de8 "(null)"
0042a3c0 int myModule!_nstream = 512
Type information missing error for _nh_malloc
004021c1 struct MyStruct myModule!MyStructInstance = struct MyStruct
00427d14 <NoType> myModule!_NLG_Destination = <no type information>
```

The x command will display an instance of a type.

```
dbgcmd

0:001> x foo!MyClassInstance
00f4f354          foo!MyClassInstance = 0x00f78768
```

The x command does not display anything based on just the name of a type.

```
dbgcmd

0:001> x foo!MyClass
0:001>
```

To display type information using the name of a type, consider using [dt \(Display Type\)](#), it provides information for both types and instances of types:

```
dbgcmd

0:001> dt foo!MyClass
+0x000 IntMemberVariable : Int4B
+0x004 FloatMemberVariable : Float
+0x008 BoolMemberVariable : Bool
+0x00c PtrMemberVariable : Ptr32 MyClass
```

Working With Templates

You can use wild cards with the x command to display template classes as shown in this sample.

```
dbgcmd
```

```
0:001> x Fabric!Common::ConfigEntry*TimeSpan?
000007f6`466a2f9c
Fabric!Common::ConfigEntry<Common::TimeSpan>::ConfigEntry<Common::TimeSpan>
(void)
000007f6`466a3020
Fabric!Common::ConfigEntry<Common::TimeSpan>::~ConfigEntry<Common::TimeSpan>
(void)
```

Consider using the [dt \(Display Type\)](#) command when working with templates, as the x command does not display individual template class items.

```
dbgcmd

0:001> dt foo!Common::ConfigEntry<Common::TimeSpan>
+0x000 __VFN_table : Ptr64
+0x008 componentConfig_ : Ptr64 Common::ComponentConfig
+0x010 section_       :
std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t>>
>
+0x038 key_           :
std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t>>
>
```

See also

[Verifying Symbols](#)

[dv \(Display Local Variables\)](#)

z (Execute While)

Article • 10/25/2023

The **z** command executes a command while a given condition is true.

User-Mode

```
dbgcmd
```

```
Command ; z( Expression )
```

Kernel-Mode

```
dbgcmd
```

```
Command ; [Processor] z( Expression )
```

Parameters

Command

Specifies the command to execute while the *Expression* condition evaluates to a nonzero value. This command is always executed at least once.

Processor

Specifies the processor that applies to the test. For more information about the syntax, see [Multiprocessor Syntax](#). You can specify processors only in kernel mode.

Expression

Specifies the condition to test. If this condition evaluates to a nonzero value, the *Command* command is executed again and then *Expression* is tested again. For more information about the syntax, see [Numerical Expression Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In many debugger commands, the semicolon is used to separate unrelated commands. However, in the **z** command, a semicolon separates the "z" from the *Command* parameter.

The *Command* command is always executed at least once, and then *Expression* is tested. If the condition is a nonzero value, the command is again executed, and then *Expression* is tested again. (This behavior is similar to a C-language **do - while** loop, not a simple **while** loop.)

If there are several semicolons to the left of the "z", all commands to the left of the "z" repeat as long as the *Expression* condition is true. Such commands can be any debugger commands that permit a terminal semicolon.

If you add another semicolon and additional commands after the **z** command, these additional commands are executed after the loop is complete. We do not typically recommend a line that begins with "z" because it generates uninteresting output forever unless the condition becomes false because of some other action. Note that you can nest **z** commands.

To break a loop that is continuing for too long, use **CTRL+C** in CDB or KD, or use **Debug | Break** from the menu, or **CTRL+BREAK** in WinDbg.

The following code example shows an unnecessarily complex way to zero the **eax** register.

```
dbgcmd  
0:000> reax = eax - 1 ; z(eax)
```

The following example increments the **eax** and **ebx** registers until one of them is at least 8 and then it increments the **ecx** register once.

```
dbgcmd  
0:000> reax=eax+1; rebx=ebx+1; z((eax<8)|(ebx<8)); recx=ecx+1
```

The following example uses C++ expression syntax and uses the pseudo-register **\$t0** as a loop variable.

```
dbgcmd
```

```
0:000> .expr /s c++
Current expression evaluator: C++ - C++ source expressions

0:000> db pindexcreate[@$t0].szKey; r$t0=@t0+1; z( @$t0 < cIndexCreate )
```

See also

[j \(Execute If-Else\)](#)

Meta-Commands

Article • 01/23/2024

This section of the reference discusses the various debugger *meta-commands* that can be used in CDB, KD, and WinDbg. These commands are preceded by a period (.).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.abandon (Abandon Process)

Article • 10/25/2023

The **.abandon** command ends the debugging session, but leaves the target application in a debugging state. This returns the debugger to dormant mode.

```
dbgcmd
```

```
.abandon [/h|/n]
```

Parameters

/h

Any outstanding debug event will be continued and marked as handled. This is the default.

/n

Any outstanding debug event will be continued unhandled.

Environment

This command is only supported in Windows XP and later versions of Windows.

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

If the target is left in a debugging state, a new debugger can be attached to it. See [Re-attaching to the Target Application](#) for details. However, after a process has been abandoned once, it can never be restored to a running state without a debugger attached.

.allow_exec_cmds (Allow Execution Commands)

Article • 10/25/2023

The `.allow_exec_cmds` command controls whether execution commands can be used.

dbgcmd

```
.allow_exec_cmds 0  
.allow_exec_cmds 1  
.allow_exec_cmds
```

Parameters

0

Prevents execution commands from being used.

1

Allows execution commands to be used.

Environment

Modes	user mode and kernel mode
Targets	live debugging only
Platforms	all

Additional Information

For a complete list of execution commands, see [Controlling the Target](#).

Remarks

With no parameters, `.allow_exec_cmds` will display whether execution commands are currently permitted.

Execution commands include [g \(Go\)](#), [t \(Trace\)](#), [p \(Step\)](#), and any other command or WinDbg graphical interface action that would cause the target to execute.

.allow_image_mapping (Allow Image Mapping)

Article • 10/25/2023

The `.allow_image_mapping` command controls whether image files will be mapped.

dbgcmd

```
.allow_image_mapping [/r] 0
.allow_image_mapping [/r] 1
.allow_image_mapping
```

Parameters

/r

Reloads all modules in the debugger's module list. This is equivalent to [.reload /d](#).

0

Prevents image files from being mapped.

1

Allows image files to be mapped.

Environment

Modes	user mode and kernel mode
Targets	live, crash dump
Platforms	all

Remarks

With no parameters, `.allow_image_mapping` will display whether image file mapping is currently allowed. By default, this mapping is allowed.

Image mapping is most common when a minidump is being debugged. Image mapping can also occur if DbgHelp is unable to access debug records (for example, during kernel debugging when memory has been paged out).

.apply_dbp (Apply Data Breakpoint to Context)

Article • 10/25/2023

The **.apply_dbp** command applies the current process' existing data breakpoints to the specified register context.

dbgcmd

```
.apply_dbp [/m Context]
```

Parameters

/m Context

Specifies the address of a register context (CONTEXT structure) in memory to which to apply the current process' data breakpoints.

Environment

Modes	user mode and kernel mode
Targets	live target only
Platforms	all

Additional Information

For more information about breakpoints controlled by the processor, see [Processor Breakpoints \(ba Breakpoints\)](#). For more information about the register context (thread context), see [Register Context](#).

Remarks

Breakpoints that are controlled by the processor are called *data breakpoints* or *processor breakpoints*. These breakpoints are created by the [ba \(Break on Access\)](#) command.

These breakpoints are associated with a memory location in the address space of a specific process. The **.apply_dbp** command modifies the specified register context so that these data breakpoints will be active when this context is used.

If the **/m Address** parameter is not used, data breakpoints will be applied to the current register context.

This command can only be used if the target is in native machine mode. For example, if the target is running on a 64-bit machine emulating an x86 processor using *WOW64*, this command cannot be used.

One example of a time this command is useful is when you are in an exception filter. The **.apply_dbp** command can update the exception filter's stored context. Data breakpoints will then be applied when the exception filter exits and the stored context is resumed. Without such a modification it is possible that data breakpoints would be lost.

.asm (Change Disassembly Options)

Article • 10/25/2023

The **.asm** command controls how disassembly code will be displayed.

```
dbgcmd  
  .asm  
  .asm[-] Options
```

Parameters

-
Causes the specified options to be disabled. If no minus sign is used, the specified options will be enabled.

Options Can be any number of the following options:

ignore_output_width

Prevents the debugger from checking the width of lines in the disassembly display.

no_code_bytes

(x86 and x64 targets only) Suppresses the display of raw bytes.

source_line

Prefaces each line of disassembly with the line number of the source code.

verbose

(Itanium target only) Causes bundle-type information to be displayed along with the standard disassembly information.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For a description of assembly debugging and related commands, see [Debugging in Assembly Mode](#).

Remarks

Using **.asm** by itself displays the current state of the options.

This command affects the display of any disassembly instructions in the Debugger Command window. In WinDbg it also changes the contents of the Disassembly window.

.attach (Attach to Process)

Article • 10/25/2023

The `.attach` command attaches to a new target application.

```
dbgcmd
```

```
.attach [-remote RemoteOptions] AttachOptions PID
```

Parameters

RemoteOptions

Specifies a process server to attach to. The options are the same as those for the command line `-remote` option. See [Activating a Smart Client](#) for syntax details.

AttachOptions

Specifies how the attach is to be done. This can include any of the following options:

-b

Prevents the debugger from requesting an initial break-in when attaching to a target process. This can be useful if the application is already suspended, or if you want to avoid creating a break-in thread in the target.

-e

Allows the debugger to attach to a process that is already being debugged. See [Re-attaching to the Target Application](#) for details.

-k

Begins a local kernel debugging session. See [Performing Local Kernel Debugging](#) for details.

-f

Freezes all threads in all target applications, except in the new target being attached to. These threads will remain frozen until an exception occurs in the newly-attached process. Note that an initial breakpoint qualifies as an exception. Individual threads can be unfrozen by using the [~u \(Unfreeze Thread\)](#) command.

-r

Causes the debugger to start the target process running when it attaches to it. This can be useful if the application is already suspended and you want it to resume execution.

-v

Causes the specified process to be debugged noninvasively.

PID

Specifies the process ID of the new target application.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

This command can be used when CDB is dormant, or if it is already debugging one or more processes. It cannot be used when WinDbg is dormant.

If this command is successful, the debugger will attach to the specified process the next time the debugger issues an execution command. If this command is used several times in a row, execution will have to be requested as many times as this command was used.

Because execution is not permitted during noninvasive debugging, the debugger is not able to noninvasively debug more than one process at a time. This also means that using the `.attach -v` command may render an already-existing invasive debugging session less useful.

Multiple target processes will always be executed together, unless some of their threads are frozen or suspended.

If you wish to attach to a new process and freeze all your existing targets, use the `-f` option. For example, you might be debugging a crash in a client application and want to attach to the server process without letting the client application continue running.

If the `-remote` option is used, the new process will be part of a new system. For details, see [Debugging Multiple Targets](#).

.beep (Speaker Beep)

Article • 10/25/2023

The **.beep** command makes noise on the computer speaker.

```
dbgcmd
```

```
  .beep
```

Environment

This command cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.bpcmds (Display Breakpoint Commands)

Article • 10/25/2023

The **.bpcmds** command displays the commands that were used to set each of the current breakpoints.

```
dbgcmd
```

```
  .bpcmds
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about and examples of how to use breakpoints, other breakpoint commands and methods of controlling breakpoints, see [Using Breakpoints](#).

Remarks

If it is unclear whether a particular breakpoint is set at an address, at a symbolic reference, or at a symbol, use the **.bpcmds** command to shows which breakpoint command was used to create it. The command that was used to create a breakpoint determines its nature:

- The [bp \(Set Breakpoint\)](#) command sets a breakpoint at an address.
- The [bu \(Set Unresolved Breakpoint\)](#) command sets a breakpoint on a symbolic reference.
- The [bm \(Set Symbol Breakpoint\)](#) command sets a breakpoint on symbols that match a specified pattern. If the **/d** switch is included, it creates zero or more

breakpoints on addresses (like **bp**), otherwise it creates zero or more breakpoints on symbolic references (like **bu**).

- The **ba (Break on Access)** command sets a data breakpoint at an address.

The output of **.bpcmds** reflects the current nature of each breakpoint. Each line of the **.bpcmds** display begins with the command used to create it (**bp**, **bu**, or **ba**) followed by the breakpoint ID, and then the location of the breakpoint.

If the breakpoint was created by **ba**, the access type and size are displayed as well.

If the breakpoint was created by **bm** without the **/d** switch, the display indicates the breakpoint type as **bu**, followed by the evaluated symbol enclosed in the **@!"** token (which indicates it is a literal symbol and not a numeric expression or register). If the breakpoint was created by **bm** with the **/d** switch, the display indicates the breakpoint type as **bp**.

Here is an example:

```
dbgcmd

0:000> bp notepad!winmain

0:000> .bpcmds
bp0 0x00000001`00003340 ;

0:000> bu myprog!winmain
breakpoint 0 redefined

0:000> .bpcmds
bu0 notepad!winmain;

0:000> bu myprog!LoadFile

0:000> bp myprog!LoadFile+10

0:000> bm myprog!openf*
 3: 00421200 @!"myprog!openFile"
 4: 00427800 @!"myprog!openFilter"

0:000> bm /d myprog!closef*
 5: 00421600 @!"myprog!closeFile"

0:000> ba r2 myprog!LoadFile+2E

0:000> .bpcmds
bu0 notepad!winmain;
bu1 notepad!LoadFile;
bp2 0x0042cc10 ;
bu3 @!"myprog!openFile";
bu4 @!"myprog!openFilter";
```

```
bp5 0x00421600 ;
ba6 r2 0x0042cc2e ;
```

In this example, notice that the output of **.bpcmds** begins with the relevant command ("bu", "bp", or "ba"), followed by the breakpoint number (with no intervening space).

Notice that because breakpoint number 0 was originally set using **bp**, and then was redefined using **bu**, the display shows its type as "bu".

Notice also that breakpoints 3, 4, and 5, which were created by the **bm** commands shown in this example, are displayed as either type "bp" or type "bu", depending on whether the **/d** switch was included when **bm** was used.

.bpsync (Synchronize Threads at Breakpoint)

Article • 10/25/2023

When a thread reaches a breakpoint, the **.bpsync** command freezes all other threads, until the thread to which the breakpoint applies has stepped through the breakpoint.

```
dbgcmd
```

```
.bpsync 1  
.bpsync 0  
.bpsync
```

Parameters

1

Causes all threads to freeze when one thread has reached a breakpoint. After the thread to which the breakpoint applies has stepped through the breakpoint, the other threads are unfrozen.

0

Allows other threads to continue executing when one thread has reached a breakpoint. This is the default behavior.

Environment

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Remarks

With no parameters, the **.bpsync** command displays the current rule governing breakpoint synchronization behavior.

The `.bpsync` command applies both to software breakpoints (set with `bp`, `bu`, or `bm`) and to processor breakpoints (set with `ba`).

If there is a possibility of multiple threads running through the same code, the `.bpsync 1` command can be useful for capturing all breakpoint occurrences. Without this command, a breakpoint occurrence could be missed because the first thread to reach the breakpoint always causes the debugger to temporarily remove the breakpoint. In the short period when the breakpoint is removed, other threads could reach the same place in the code and not trigger the breakpoint as intended.

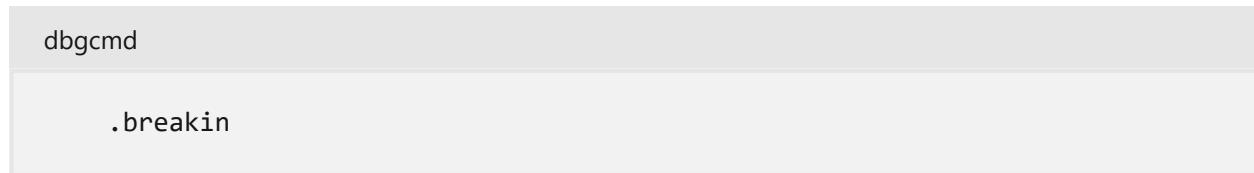
The temporary removal of breakpoints is a normal aspect of debugger operation. When the target reaches a breakpoint and is resumed, the debugger has to remove the breakpoint temporarily so that the target can execute the real code. After the real instruction has been executed, the debugger reinserts the break. To do this, the debugger restores the code (or turns off data breaks), does a single-step, and then puts the break back.

Note that if you use `.bpsync 1`, there is a risk of deadlocks among the threads that have been frozen.

.breakin (Break to the Kernel Debugger)

Article • 10/25/2023

The **.breakin** command switches from user-mode debugging to kernel-mode debugging. This command is particularly useful when you are controlling the user-mode debugger from the kernel debugger.



Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

If kernel-mode debugging was enabled during the boot process and you are running a user-mode debugger, you can use the **.breakin** command to halt the operating system and transfer control to a kernel debugger.

The **.breakin** command causes a kernel-mode break in the debugger's process context. If a kernel debugger is attached, it will become active. The kernel debugger's [process context](#) will automatically be set to the process of the user-mode debugger, not the user-mode debugger's target process.

This command is primarily useful when debugging a user-mode problem requires retrieving information about the kernel state of the system. Resuming execution in the kernel debugger is necessary before the user-mode debugging session can continue.

When you are [controlling the user-mode debugger from the kernel debugger](#) and the user-mode debugger prompt is visible in the kernel debugger, this command will pause the user-mode debugger and make the kernel-mode debugging prompt appear.

If the system is unable to break into the kernel debugger, an error message is displayed.

This command is also useful if you use the kernel debugger to set a breakpoint in user space and that breakpoint is caught by a user-mode debugger instead of the kernel debugger. Issuing this command in the user-mode debugger will transfer control to the kernel debugger.

If the **.breakin** command is used on a system that was not booted with debugging enabled, it has no effect.

.browse (Display Command in Browser)

Article • 10/25/2023

The `.browse` command displays the output of a specified command in a new [Command Browser window](#).

```
dbgcmd
  .browse Command
```

Parameters

Command

The command to be executed and displayed in a new Command Browser window.

Remarks

The following example uses the `.browse` command to display the output of the [.chain /D](#) command in a Command Browser window.

```
dbgcmd
  .browse .chain /D
```

.bugcheck (Display Bug Check Data)

Article • 04/03/2024

The `.bugcheck` command displays the data from a bug check on the target computer.

dbgsyntax

`.bugcheck`

Environment

[+] Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about bug checks, see [Bug Checks \(Blue Screens\)](#). For a description of individual bug checks, see the [Bug Check Code Reference](#) section.

Remarks

This command displays the current bug check data. (This bug check data will be accessible until the crashed machine is rebooted.)

You can also display bug check data on 32-bit systems by using `dd NT!KiBugCheckData L5`, or on 64-bit systems by using `dq NT!KiBugCheckData L5`. However, the `.bugcheck` command is more reliable, because it works in some scenarios that the [d* \(Display Memory\)](#) command will not (such as user-mode minidumps).

The [!analyze](#) extension command is also useful after a bug check occurs.

Feedback

Was this page helpful?

 Yes

 No

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

.cache (Set Cache Size)

Article • 10/25/2023

The `.cache` command sets the size of the cache used to hold data obtained from the target. Also sets a number of cache and memory options.

dbgsyntax

```
.cache Size  
.cache Option  
.cache
```

Parameters

Size

The size of the kernel debugging cache, in kilobytes. If *Size* is zero, the cache is disabled. The command output displays the cache size in bytes. (The default size is 1000 KB.)

Option

Can be any one of the following options:

hold

Automatic cache flushing is disabled.

unhold

Turns off the **hold** option. (This is the default setting.)

decodeptes

All transition page table entries (PTEs) will be implicitly decoded. (This is the default setting.)

nodelcodeptes

Turns off the **decodeptes** option.

forcedecodeptes

All virtual addresses will be translated into physical addresses before access. This option also causes the cache to be disabled. Unless you are concerned with kernel-mode memory, it is more efficient to use **forcedecodeuser** instead.

forcedecodeuser

All user-mode virtual addresses will be translated into physical addresses before access. This option also causes the cache to be disabled.

Note You must activate **forcedecodeuser** (or **forcedecodeptes**) before using [.thread \(Set Register Context\)](#), [.context \(Set User-Mode Address Context\)](#), [.process \(Set Process Context\)](#), or [!session](#) during live debugging. If you use the /p option with **.thread** and **.process**, the **forcedecodeuser** option is automatically set. In any other case, you will need to use the **.cache forcedecodeuser** command explicitly.

noforcedecodeptes

Turns off the **forcedecodeptes** and **forcedecodeuser** options. (This is the default setting.)

flushall

Deletes the entire virtual memory cache.

flushu

Deletes all entries of ranges with errors from the cache, as well as all user-mode entries.

flush Address

Deletes a 4096-byte block of the cache, beginning at *Address*.

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Remarks

If **.cache** is used with no arguments, the current cache size, status, and options are displayed.

The **.cache forcedecodeuser** or **.cache forcedecodeptes** option will only last as long as the debugger remains broken into the target computer. If any stepping or execution of the target takes place, the **noforcedecodeptes** state will again take effect. This prevents the debugger from interfering with execution or a reboot in an unproductive manner.

.call (Call Function)

Article • 10/25/2023

The `.call` command causes the target process to execute a function.

dbgsyntax

```
.call [/v] Function( Arguments )
.call /s Prototype Function( Arguments )
.call /c
.call /C
```

Parameters

/v

Verbose information about the call and its arguments is displayed.

/s *Prototype*

Allows you to call the function that is specified by *Function* even though you do not have the correct symbols. In this case, you must have symbols for another function that has the same calling prototype as the function you are trying to call. The *Prototype* parameter is the name of this prototype function.

Function

Specifies the function being called. This can be the name of the function (preferably qualified with a module name), or any other expression that evaluates to the function address. If you need to call a constructor or destructor, you must supply the address -- or else use a C++ expression to evaluate named syntax for the operators (see [Numerical Expression Syntax](#) for details).

Arguments

Specifies the arguments passed to the function. If you are calling a method, the first argument must be `this`, and all other arguments follow it. Arguments should be separated with commas and should match the usual argument syntax. Variable-length argument lists are supported. Expressions within an argument are parsed by the C++ expression evaluator; see [C++ Numbers and Operators](#) for details. You cannot enter a literal string as an argument, but you can use a pointer to a string, or any other memory accessible to the target process.

/c

Clears any existing call on the current thread.

/C

Clears any existing call on the current thread, and resets the context of the current thread to the context stored by the outstanding call.

Environment

Modes	user mode only
Targets	live debugging only
Platforms	x86 and x64 only

Remarks

The specified function is called by the current thread of the current process.

Only the **cdecl**, **stdcall**, **fastcall**, and **thiscall** calling conventions are supported. Managed code cannot be called by this command.

After **.call** is used, the debugger will update the stack, change the instruction pointer to point to the beginning of the called function, and then stop. Use [g \(Go\)](#) to resume execution, or [~. g](#) to execute just the thread making the call.

When the function returns, a break occurs and the debugger displays the return value of the function. The return value is also stored in the **\$callret** pseudo-register, which acquires the type of the return value.

If you have broken into the target using CTRL+C or CTRL+BREAK, the current thread is an additional thread created to handle the breakin. If you issue a **.call** command at this point, the extra thread will be used for the called function.

If you have reached a predefined breakpoint, there is no extra breakin thread. If you use **.call** while at a breakpoint in user mode, you could use [g](#) to execute the entire process, or [~. g](#) to execute just the current thread. Using [g](#) may distort your program's behavior, since you have taken one thread and diverted it to this new function. On the other hand, this thread will still have its locks and other attributes, and thus [~. g](#) may risk deadlocks.

The safest way to use **.call** is to set a breakpoint in your code at a location where a certain function could be safely called. When that breakpoint is hit, you can use **.call** if you desire that function to run. If you use **.call** at a point where this function could not normally be called, a deadlock or target corruption could result.

It may be useful to add extra functions to your source code that are not called by the existing code, but are intended to be called by the debugger. For example, you could add functions that are used to investigate the current state of your code and its environment and store information about the state in a known memory location. Be sure not to optimize your code, or these functions may be removed by the compiler. Use this technique only as a last resort, because if your application crashes `.call` will not be available when debugging the dump file.

The `.call /c` and `.call /C` commands should only be used if an attempt to use `.call` has failed, or if you changed your mind before entering the `g` command. These should not be used casually, since abandoning an uncompleted call can lead to a corrupted target state.

The following code example shows how the `.call /s` command is used.

```
dbgcmd  
.call /s KnownFunction UnknownFunction( 1 )
```

In this example, you have private symbols for `KnownFunction`, which takes an integer as its only argument and returns, for example, a pointer to an array. You do not have symbols, or possibly you only have public symbols for `UnknownFunction`, but you do know that it takes an integer as its only argument and returns a pointer to an array. By using the `/s` option, you can specify that `UnknownFunction` will work the same way that `KnownFunction` does. Thus, you can successfully generate a call to `UnknownFunction`.

.chain (List Debugger Extensions)

Article • 10/25/2023

The **.chain** command lists all loaded debugger extensions in their default search order.

dbgsyntax

```
.chain  
.chain /D
```

Parameters

/D

Displays the output using [Debugger Markup Language](#). In the output, each listed module is a link that you can click to get information about the extensions that are implemented by the module.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details on loading, unloading, and controlling extensions, see [Loading Debugger Extension DLLs](#). For details on executing extension commands and an explanation of the default search order, see [Using Debugger Extension Commands](#).

.childdbg (Debug Child Processes)

Article • 10/25/2023

The `.childdbg` command controls the debugging of child processes.

dbgsyntax

```
.childdbg 0  
.childdbg 1  
.childdbg
```

Parameters

0

Prevents the debugger from debugging child processes.

1

Causes the debugger to debug child processes.

Environment

This command is only supported in Windows XP and later versions of Windows.

Modes	user mode only
Targets	live debugging only
Platforms	x86 and x64

Remarks

Child processes are additional processes launched by the original target application.

With no parameters, `.childdbg` will display the current status of child-process debugging.

.clients (List Debugging Clients)

Article • 10/25/2023

The `.clients` command lists all debugging clients currently connected to the debugging session.

dbgsyntax

```
.clients
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more details and other commands that can be used while performing remote debugging through the debugger, see [Controlling a Remote Debugging Session](#).

.closehandle (Close Handle)

Article • 10/25/2023

The **.closehandle** command closes a handle owned by the target application.

dbgsyntax

```
.closehandle Handle  
.closehandle -a
```

Parameters

Handle

Specifies the handle to be closed.

-a

Causes all handles owned by the target application to be closed.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

You can use the [!handle](#) extension to display the existing handles.

.cls (Clear Screen)

Article • 10/25/2023

The `.cls` command clears the Debugger Command window display.

dbgsyntax

`.cls`

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.context (Set User-Mode Address Context)

Article • 10/25/2023

The **.context** command specifies which page directory of a process will be used for the user-mode address context, or displays the current user-mode address context.

dbgsyntax

```
.context [PageDirectoryBase]
```

Parameters

PageDirectoryBase

Specifies the base address for a page directory of a desired process. The user-mode address context will be set to this page directory. If *PageDirectoryBase* is zero, the user-mode address context will be set to the page directory for the current system state. If *PageDirectoryBase* is omitted, the current user-mode address context is displayed.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the user-mode address context and other context settings, see [Changing Contexts](#).

Remarks

Generally, when you are doing kernel debugging, the only visible user-mode address space is the one associated with the current process.

The **.context** command instructs the kernel debugger to use the specified page directory as the *user-mode address context*. After this command is executed, the

debugger will have access to this virtual address space. The page tables for this address space will be used to interpret all user-mode memory addresses. This allows you to read and write to this memory.

The [.process \(Set Process Context\)](#) command has a similar effect. However, the [.context](#) command sets the user-mode address context to a specific page directory, while the [.process](#) command sets the *process context* to a specific process. On an x86 processor, these two commands have essentially the same effect. See [Process Context](#) for more details.

If you are doing live debugging, you should issue a [.cache forcedecodeuser](#) command in addition to the [.context](#) command. This forces the debugger to look up the physical addresses of the memory space needed. (This can be slow, because it often means a huge amount of data must be transferred across the debug cable.)

If you are doing crash dump debugging, the [.cache](#) command is not needed. However, you will not have access to any portions of the virtual address space of the user-mode process that were paged to disk when the crash occurred.

Here is an example. Use the [!process](#) extension to find the directory base for the desired process:

```
dbgcmd

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0 SessionId: 0 Cid: 0008 Peb: 00000000 ParentCid: 0000
    DirBase: 00030000 ObjectTable: fe529b68 TableSize: 50.
    Image: System

...
PROCESS fe3c0d60 SessionId: 0 Cid: 0208 Peb: 7ffdf000 ParentCid: 00d4
    DirBase: 0011f000 ObjectTable: fe3d0f48 TableSize: 30.
    Image: regsvc.exe
```

Now use the [.context](#) command with this page directory base.

```
dbgcmd

kd> .context 0011f000
```

This enables you to examine the address space in various ways. For example, here is the output of the [!peb](#) extension:

```
dbgcmd
```

```
kd> !peb
PEB at 7FFDF000
  InheritedAddressSpace:      No
  ReadImageFileExecOptions:   No
  BeingDebugged:             No
  ImageBaseAddress:          01000000
  Ldr.Initialized:           Yes
  Ldr.InInitializationOrderModuleList: 71f40 . 77f68
  Ldr.InLoadOrderModuleList:  71ec0 . 77f58
  Ldr.InMemoryOrderModuleList: 71ec8 . 77f60
    01000000 C:\WINNT\system32\regsvc.exe
    77F80000 C:\WINNT\System32\ntdll.dll
    77DB0000 C:\WINNT\system32\ADVAPI32.dll
    77E80000 C:\WINNT\system32\KERNEL32.DLL
    77D40000 C:\WINNT\system32\RPCRT4.DLL
    77BE0000 C:\WINNT\system32\secur32.dll
  SubSystemData:              0
  ProcessHeap:                70000
  ProcessParameters:          20000
    WindowTitle:   'C:\WINNT\system32\regsvc.exe'
    ImageFile:     'C:\WINNT\system32\regsvc.exe'
    CommandLine:  'C:\WINNT\system32\regsvc.exe'
    DllPath:
'C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\sys
tem32;C:\WINNT;C:\WINNT\System32\Wbem;C:\PROGRA~1\COMMON~1\AUTODE~1'
  Environment:   0x10000
```

.copysym (Copy Symbol Files)

Article • 10/25/2023

The **.copysym** command copies the current symbol files to the specified directory.

dbgsyntax

```
.copysym [/l] Path
```

Parameters

/l

Causes each symbol file to be loaded as it is copied.

Path

Specifies the directory to which the symbol files should be copied. Copies do not overwrite existing files.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Many times, symbols are stored on a network. The symbol access can often be slow, or you may need to transport your debugging session to another computer where you no longer have network access. In such scenarios, the **.copysym** command can be used to copy the symbols you need for your session to a local directory.

.cordll (Control CLR Debugging)

Article • 10/25/2023

The **.cordll** command controls managed code debugging and the Microsoft .NET common language runtime (CLR).

dbgsyntax

```
.cordll [Options]
```

Parameters

Options One or more of the following options:

-I (lower-case L)

Loads the CLR debugging modules.

-I ** *Module*** (upper-case i)

Specifies the name or base address of the CLR module to be debugged. For more information, see Remarks.

-u

Unloads the CLR debugging modules.

-e

Enables CLR debugging.

-d

Disables CLR debugging.

-D

Disables CLR debugging and unloads the CLR debugging modules.

-N

Reloads the CLR debugging modules.

-Ip ** *Path***

Specifies the directory path of the CLR debugging modules.

-se

Enables using the short name of the CLR debugging module, mscordacwks.dll.

-sd

Disables using the short name of the CLR debugging module, msordacwks.dll. Instead, the debugger uses the long name of the CLR debugging module, msordacwks_<spec>.dll. Turning off short name usage enables you to avoid having your local CLR used if you are concerned about mismatches.

-ve

Turns on verbose mode for CLR module loading.

-vd

Turns off verbose mode for CLR module loading.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

To debug a managed application, the debugger must load a data access component (DAC) that corresponds to the CLR that the application has loaded. However, in some cases, the application loads more than one CLR. In that case, you can use the **I** parameter to specify which DAC the debugger should load. Version 2 of the CLR is named Mscorwks.dll, and version 4 of the CLR is named Clr.dll. The following example shows how to specify that the debugger should load the DAC for version 2 (mscorwks).

```
dbgcmd  
.cordll -I mscorwks -lp c:\dacFolder
```

If you omit the **I** parameter, the debugger uses version 4 by default. For example, the following two commands are equivalent.

```
dbgcmd  
.cordll -lp c:\dacFolder  
.cordll -I clr -lp c:\dacFolder
```

Sos.dll is a component that is used for debugging managed code. The current version of Debugging Tools for Windows does not include any version of sos.dll. For information about how to get sos.dll, see *Getting the SOS Debugging Extension (sos.dll)* in [Debugging Managed Code Using the Windows Debugger](#).

The `.cordll` command is supported in kernel-mode debugging. However, this command might not work unless the necessary memory is paged in.

See also

[Debugging Managed Code Using the Windows Debugger](#)

[SOS Debugging Extension](#)

.crash (Force System Crash)

Article • 10/25/2023

The **.crash** command causes the target computer to issue a bug check.

dbgsyntax

.crash

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

For an overview of related commands and a description of the options available after a system crash, see [Crashing and Rebooting the Target Computer](#).

Remarks

This command will immediately cause the target computer to crash.

If you are already in a bug check handler, do not use **.crash**. Use [**g \(Go\)**](#) instead to continue execution of the handler, which will generate a crash dump.

A kernel-mode dump file will be written if crash dumps have been enabled. See [Creating a Kernel-Mode Dump File](#) for details.

.create (Create Process)

Article • 10/25/2023

The `.create` command creates a new target application.

dbgsyntax

```
.create [-remote RemoteOptions] [-f] CommandLine
```

Parameters

RemoteOptions

Specifies a process server to which to attach. The options are the same as those for the command line `-remote` option. See [Activating a Smart Client](#) for syntax details.

`-f`

Freezes all threads in all target applications, except in the new target being created.

These threads will remain frozen until an exception occurs in the newly-created process.

Note that an initial breakpoint qualifies as an exception. Individual threads can be unfrozen by using the [~u \(Unfreeze Thread\)](#) command.

CommandLine

Specifies the complete command line for the new process. *CommandLine* may contain spaces, and must not be surrounded with quotes. All text after the `.create` command is taken as part of *CommandLine*; this command cannot be followed with a semicolon and additional debugger commands.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

This command can be used when CDB is dormant, or if it is already debugging one or more processes. It cannot be used when WinDbg is dormant.

If this command is successful, the debugger will create the specified process the next time the debugger issues an execution command. If this command is used several times in a row, execution will have to be requested as many times as this command was used.

Multiple target processes will always be executed together, unless some of their threads are frozen or suspended.

If you wish to create a new process and freeze all your existing targets, use the `-f` option.

If the `-premove` option is used, the new process will be part of a new system. For details, see [Debugging Multiple Targets](#).

.createdir (Set Created Process Directory)

Article • 10/25/2023

The **.createdir** command controls the starting directory and handle inheritance for any processes created by the debugger.

dbgsyntax

```
.createdir [-i | -I] [Path]
```

Parameters

-i

Causes processes created by the debugger to inherit handles from the debugger. This is the default.

-I

Prevents processes created by the debugger from inheriting handles from the debugger.

Path

Specifies the starting directory for all child processes created by any target process. If *Path* contains spaces, it must be enclosed in quotation marks.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

If **.createdir** is used with no parameters, the current starting directory and handle inheritance status are displayed.

If `.createdir` has never been used, any created process will use its usual default directory as its starting directory. If you have already set a path with `.createdir` and want to return to the default status, use `.createdir ""` with nothing inside the quotation marks.

The `.createdir` setting affects all processes created by [.create \(Create Process\)](#). It also affects processes created by WinDbg's **File | Open Executable** menu command, unless the **Start directory** text box is used to override this setting.

.cxr (Display Context Record)

Article • 03/07/2024

The `.cxr` command displays the context record saved at the specified address. It also sets the register context.

dbgsyntax

```
.cxr [Options] [Address]
```

Parameters

Options - Can be any combination of the following options.

/f Size

Forces the context size to equal the value of *Size*, in bytes. This can be useful when the context does not match the actual target -- for example, when using an x86 context on a 64-bit target during *WOW64* debugging. If an invalid or inconsistent size is specified, the error "Unable to convert context to canonical form" will be displayed.

/w

Writes the current context to memory, and displays the address of the location where it was written.

Address - Address of the system context record (CONTEXT structure) to be read/written. An address of -1, causes the CONTEXT structure to be read from the dump file. This is the same behavior as [.ecxr](#). An address of 0 is not supported. When debugging a dump file, omitting the address resets the register context to the default context for that thread (User Mode) or processor (Kernel Mode).

Environment

[+] Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the register context and other context settings, see [Changing Contexts](#).

Remarks

The information from a context record can be used to assist in debugging a system halt where an unhandled exception has occurred and an exact stack trace is not available. The `.cxr` command displays the important registers for the specified context record.

This command also instructs the debugger to use the specified context record as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context persists until you allow the target to execute or use another register context command ([.thread](#), [.ecxr](#), [.trap](#), or `.cxr` again). In user mode, it will also be reset if you change the current process or thread. See [Register Context](#) for details.

The `.cxr` command is often used to debug bug check 0x1E. For more information and an example, see [Bug Check 0x1E \(KMODE_EXCEPTION_NOT_HANDLED\)](#).

The `.cxr /w` command writes the context to memory and displays the address where it has been stored. This address can be passed to [.apply_dbp \(Apply Data Breakpoint to Context\)](#) if you need to apply data breakpoints to this context.

See also

[Changing Contexts](#)

[Register Context](#)

[.exr \(Display Exception Record\)](#)

[.ecxr](#)

[.trap](#)

.dbgdbg (Debug Current Debugger)

Article • 10/25/2023

The **.dbgdbg** command launches a new instance of CDB; this new debugger takes the current debugger as its target.

```
dbgcmd
```

```
.dbgdbg
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.dbgdbg** command is similar to the [CTRL+P \(Debug Current Debugger\)](#) control key. However, **.dbgdbg** is more versatile, because it can be used from WinDbg as well as KD and CDB, and it can be used to debug a debugging server on a remote computer.

.detach (Detach from Process)

Article • 10/25/2023

The **.detach** command ends the debugging session, but leaves any user-mode target application running.

```
dbgcmd
```

```
.detach [ /h | /n ]
```

Parameters

/h

Any outstanding debug event will be continued and marked as handled. This is the default.

/n

Any outstanding debug event will be continued without being marked as handled.

Environment

For live user-mode debugging, this command is only supported in Windows XP and later versions of Windows.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command will end the debugging session in any of the following scenarios:

- When you are debugging a user-mode or kernel-mode dump file.
- When you are debugging a live user-mode target.
- When you are noninvasively debugging a user-mode target.

If you are only debugging a single target, the debugger will return to dormant mode after it detaches.

If you are [debugging multiple targets](#), the debugging session will continue with the remaining targets.

.dml_flow (Unassemble with Links)

Article • 10/25/2023

The `.dml_flow` command displays a disassembled code block and provides links that you can use to construct a code flow graph.

```
dbgcmd  
.dml_flow Start Target
```

Parameters

Start

The address of an instruction from which the target address can be reached.

Target

An address in the code block to be disassembled.

Remarks

Consider the call stack shown in the following example.

```
dbgcmd  
0: kd> kL  
Child-SP          RetAddr          Call Site  
fffff880`0335c688  fffff800`01b41f1c  nt!IoCallDriver  
fffff880`0335c690  fffff800`01b3b6b4  nt!IoSynchronousPageWrite+0x1cc  
fffff880`0335c700  fffff800`01b4195e  nt!MiFlushSectionInternal+0x9b8  
...
```

Suppose you want to examine all code paths from the start of `nt!MiFlushSectionInternal` to the code block that contains the return address, `fffff800`01b3b6b4`. The following command gets you started.

```
dbgcmd  
.browse .dml_flow nt!MiFlushSectionInternal fffff800`01b3b6b4
```

The output, in the [Command Browser window](#), is shown in the following image.

The screenshot shows a WinDbg window with the command `.dml_flow nt!MiFlushSectionInternal fffff800`01b3b6b4` entered in the Command field. At the top, there is a link labeled `fffff800`01b3b681`. The assembly code displayed is:

```
nt!MiFlushSectionInternal+0x997 (fffff800`01b3b693):
d:\...\winkernel\wtos\mm\flushsec.c
2934 fffff800`01b3b693 mov rsi,qword ptr [rbp-50h]
2934 fffff800`01b3b697 mov rcx,qword ptr [rbp-18h]
2934 fffff800`01b3b69b lea r9,[rbp]
2934 fffff800`01b3b69f lea r8,[rbp+30h]
2934 fffff800`01b3b6a3 mov rdx,rbx |
2934 fffff800`01b3b6a6 mov qword ptr [rsp+20h],rsi
2928 fffff800`01b3b6ab mov dword ptr [rbp+4],r14d
2934 fffff800`01b3b6af call nt!IoSynchronousPageWrite (fffff
2936 fffff800`01b3b6b4 test eax,eax
2936 fffff800`01b3b6b6 js nt!MiFlushSectionInternal+0xe8f
```

At the bottom of the assembly code, there are two links: `fffff800`01b3b6bc` and `fffff800`01b3bb87`.

The preceding image shows the code block that contains the target address, `fffff800`01b3b6b4`. There is only one link (`fffff800`01b3b681`) at the top of the image. That indicates that there is only one code block from which the current code block can be reached. If you click the link, you will see that code block disassembled, and you will see links that enable you to further explore the code flow graph.

The two links at the bottom of the preceding image indicate that there are two code blocks that can be reached from the current code block.

See also

[Debugger Markup Language Commands](#)

[uf \(Unassemble Function\)](#)

.dml_start (Display DML Starting Point)

Article • 10/25/2023

The **.dml_start** command displays output that serves as a starting point for exploration using commands that support [Debugger Markup Language \(DML\)](#).

```
dbgcmd

.dml_start
.dml_start filename
```

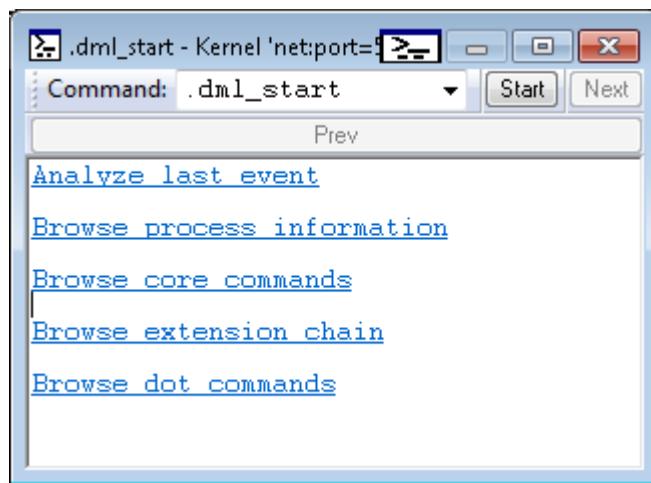
Parameters

filename

The name of a DML file to be displayed as the starting output.

Using the Default Starting Output

If *filename* is omitted, the debugger displays a default DML starting output as illustrated in the following image.



Each line of output in the preceding example is a link that you can click to invoke other commands.

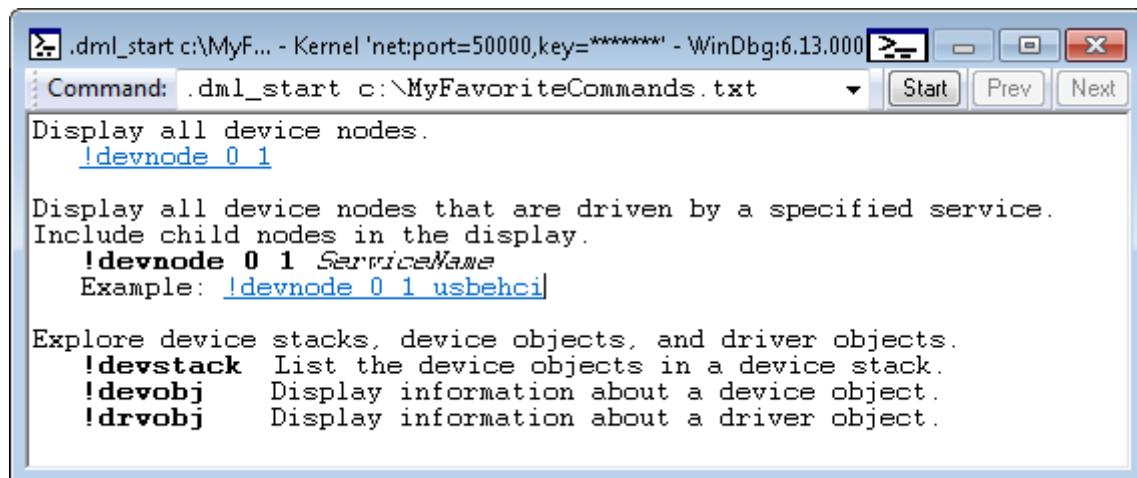
Providing a DML File

If you supply a path to a DML file, the file is used as the starting output. For example, suppose the file c:\MyFavoriteCommands.txt contains the following text and DML tags.

dbgcmd

```
Display all device nodes.  
<link cmd="!devnode 0 1">!devnode 0 1</link>  
  
Display all device nodes that are driven by a specified service.  
Include child nodes in the display.  
<b>!devnode 0 1</b> <i>ServiceName</i>  
Example: <link cmd="!devnode 0 1 usbehci">!devnode 0 1 usbehci</link>  
  
Explore device stacks, device objects, and driver objects.  
<b>!devstack</b> List the device objects in a device stack.  
<b>!devobj</b> Display information about a device object.  
<b>!drvobj</b> Display information about a driver object.
```

The command `.dml_start c:\MyFavoriteCommands.txt` will display the file as shown in the following image.



Remarks

For information about DML tags that can be used in DML files, see `dml.doc` in the installation folder for Debugging Tools for Windows.

DML output often works well in the [Command Browser window](#). To display a DML file in the Command Browser window, use `.browse .dml_start filename`.

See also

[Debugger Markup Language Commands](#)

[.browse](#)

.dump (Create Dump File)

Article • 10/25/2023

The `.dump` command creates a user-mode or kernel-mode crash dump file.

```
dbgcmd  
.dump [options] FileName  
.dump /?
```

Parameters

Options

Represents one or more of the following options.

/a

Create dumps for all processes (requires `-u`).

/b[a]

Package dump in a CAB and delete dump. Additional information is included if the `a` option is specified.

/c <comment>

Add a comment (not supported in all formats).

/j <addr>

Provide a JIT_DEBUG_INFO address.

/o

Overwrites an existing dump file with the same name. If this option is not used and there is a file with the same file name, the dump file is not written.

/u

Append unique identifier to dump name.

/f[FullOptions]

(Kernel mode:) Creates a [complete memory dump](#).

(User mode:) Not supported. Previously this option would create a *full user-mode dump*, which is a legacy format with less information than the newer minidump format. For more information, see [Varieties of User-Mode Dump Files](#).

You can add the following *FullOptions* to change the contents of the dump file; the option is case-sensitive.

FullOption	Effect
y	Adds AVX register information to the dump file.

/m[*MiniOptions*]

Creates a *small memory dump* (in kernel mode) or a *minidump* (in user mode). For more information, see [User-Mode Dump Files](#). If neither /f nor /m is specified, /m is the default.

In user mode, /m can be followed with additional *MiniOptions* specifying extra data that is to be included in the dump. If no *MiniOptions* are included, the dump will include module, thread, and stack information, but no additional data. You can add any of the following *MiniOptions* to change the contents of the dump file; they are case-sensitive.

MiniOption	Effect
a	Creates a minidump with all optional additions. The /ma option is equivalent to /mfFhut -- it adds full memory data, handle data, unloaded module information, basic memory information, and thread time information to the minidump. Any failure to read inaccessible memory results in termination of the minidump generation.
A	The /mA option is equivalent to /ma except that it ignores any failure to read inaccessible memory and continues generating the minidump.
f	Adds full memory data to the minidump. All accessible committed pages owned by the target application will be included.
F	Adds all basic memory information to the minidump. This adds a stream to the minidump that contains all basic memory information, not just information about valid memory. This allows the debugger to reconstruct the complete virtual memory layout of the process when the minidump is being debugged.
h	Adds data about the handles associated with the target application to the minidump.
u	Adds unloaded module information to the minidump.
t	Adds additional thread information to the minidump. This includes thread times, which can be displayed by using the !runaway extension or the .ttime (Display Thread Times) command when debugging the minidump.
i	Adds secondary memory to the minidump. Secondary memory is any memory referenced by a pointer on the stack or backing store, plus a small region surrounding this address.

MinOption	Effect
p	Adds process environment block (PEB) and thread environment block (TEB) data to the minidump. This can be useful if you need access to Windows system information regarding the application's processes and threads.
w	Adds all committed read-write private pages to the minidump.
d	Adds all read-write data segments within the executable image to the minidump.
c	Adds code sections within images.
r	Deletes from the minidump those portions of the stack and store memory that are not useful for recreating the stack trace. Local variables and other data type values are deleted as well. This option does not make the minidump smaller (because these memory sections are simply zeroed), but it is useful if you want to protect the privacy of other applications.
R	Deletes the full module paths from the minidump. Only the module names will be included. This is a useful option if you want to protect the privacy of the user's directory structure.
y	Adds AVX register information to the dump file.

Kernel Mode Options

The following options are available in kernel mode.

/k

Create a dump with kernel memory only.

/ka

Create a dump with active kernel and user mode memory.

Additional Information

For a description of kernel-mode dump files and an explanation of their use, see [Kernel-Mode Dump Files](#). For a description of user-mode dump files and an explanation of their use, see [User-Mode Dump Files](#).

Remarks

This command can be used in a variety of situations:

- During live user-mode debugging, this command directs the target application to generate a dump file, but the target application does not terminate.
- During live kernel-mode debugging, this command directs the target computer to generate a dump file, but the target computer does not crash.
- During crash dump debugging, this command creates a new crash dump file from the old one. This is useful if you have a large crash dump file and want to create a smaller one.

You can control what type of dump file will be produced:

- In kernel mode, to produce a [complete memory dump](#), use the `/f` option. To produce a [small memory dump](#), use the `/m` option (or no options). The `.dump` command cannot produce a [kernel memory dump](#).
- In user mode, `.dump /m[MiniOptions]` is the best choice. Although "m" stands for "minidump", the dump files created by using this *MiniOption* can vary in size from very small to very large. By specifying the proper *MiniOptions* you can control exactly what information is included. For example, `.dump /ma` produces a dump with a great deal of information. The older command, `.dump /f`, produces a moderately large "standard dump" file and cannot be customized.

You cannot specify which process is dumped. All running processes will be dumped.

The `/xc`, `/xr`, `/xp`, and `/xt` options are used to store exception and context information in the dump file. This allows the [.ecxr \(Display Exception Context Record\)](#) command to be run on this dump file.

The following example will create a user-mode minidump, containing full memory and handle information:

```
dbgcmd  
0:000> .dump /mfh myfile.dmp
```

Handle information can be read by using the [!handle](#) extension command.

See Also

[Varieties of Kernel-Mode Dump Files](#)

.dumpcab (Create Dump File CAB)

Article • 10/25/2023

The **.dumpcab** command creates a CAB file containing the current dump file.

```
dbgcmd  
.dumpcab [-a] CabName
```

Parameters

-a

Causes all currently loaded symbols to be included in the CAB file. For minidumps, all loaded images will be included as well. Use [!ml](#) to determine which symbols and images are loaded.

CabName

The CAB file name, including extension. *CabName* can include an absolute or relative path; relative paths are relative to the directory in which the debugger was started. It is recommended that you choose the extension .cab.

Environment

Modes	user mode, kernel mode
Targets	crash dump
Platforms	all

Additional Information

For more details on crash dumps, see [Crash Dump Files](#).

Remarks

This command can only be used if you are already debugging a dump file.

If you are debugging a live target and want to create a dump file and place it in a CAB, you should use the [.dump \(Create Dump File\)](#) command. Next, start a new debugging session with the dump file as its target, and use [.dumpcab](#).

The **.dumpcab** command cannot be used to place multiple dump files into one CAB file.

.dvalloc (Allocate Memory)

Article • 10/25/2023

The `.dvalloc` command causes Windows to allocate additional memory to the target process.

dbgcmd

```
.dvalloc [Options] Size
```

Parameters

Options Can be any number of the following options:

/b ** BaseAddress**

Specifies the virtual address of the beginning of the allocation.

/r

Reserves the memory in the virtual address space but does not actually allocate any physical memory. If this option is used, the debugger calls `VirtualAllocEx` with the *fAllocationType* parameter equal to `MEM_RESERVE`. If this option is not used, the value `MEM_COMMIT | MEM_RESERVE` is used. See the Microsoft Windows SDK for details.

Size

Specifies the amount of memory to be allocated, in bytes. The amount of memory available to the program will equal *Size*. The amount of memory actually used may be slightly larger, since it is always a whole number of pages.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

The `.dvalloc` command calls `VirtualAllocEx` to allocate new memory for the target process. The allocated memory permits reading, writing, and execution.

To free this memory, use [`.dvfree \(Free Memory\)`](#).

.dvfree (Free Memory)

Article • 10/25/2023

The `.dvfree` command frees a memory allocation owned by the target process.

dbgcmd

```
.dvfree [/d] BaseAddress Size
```

Parameters

/d

Decommits the allocation, but does not actually release the pages containing the allocation. If this option is used, the debugger calls `VirtualFreeEx` with the `dwFreeType` parameter equal to `MEM_DECOMMIT`. If this option is not used, the value `MEM_RELEASE` is used. See the Microsoft Windows SDK for details.

BaseAddress

Specifies the virtual address of the beginning of the allocation.

Size

Specifies the amount of memory to be freed, in bytes. The actual memory freed will always be a whole number of memory pages.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Remarks

The `.dvfree` command calls `VirtualFreeEx` to free an existing memory allocation. Unless the `/d` option is specified, the pages containing this memory are released.

This command can be used to free an allocation made by [.dvalloc \(Allocate Memory\)](#). It can also be used to free any block of memory owned by the target process, but freeing

memory that was not acquired through `.dvalloc` will naturally pose risks to the stability of the target process.

.echo (Echo Comment)

Article • 10/25/2023

The **.echo** command displays a comment string.

dbgcmd

```
.echo String  
.echo "String"
```

Parameters

String

Specifies the text to display. You can also enclose *String* in quotation marks (""). Regardless of whether you use quotation marks, *String* can contain any number of spaces, commas, and single quotation marks (''). If you enclose *String* in quotation marks, it can include semicolons, but not additional quotation marks. If you do not enclose *String* in quotation marks, it can include quotation marks in any location except the first character, but it cannot include semicolons.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.echo** command causes the debugger to display *String* immediately after you enter the command.

An **.echo** command is ended if the debugger encounters a semicolon (unless the semicolon occurs within a quoted string). This restriction enables you to use **.echo** in more complex constructions such as with the [j \(Execute If - Else\)](#) command, as the following example shows.

dbgcmd

```
0:000> j (poi(MyVar)>5) '.echo MyVar Too Big'; '.echo MyVar Acceptable;
```

The **.echo** command also provides an easy way for users of debugging servers and debugging clients to communicate with one another. For more information about this situation, see [Controlling a Remote Debugging Session](#).

The **.echo** command differs from the [\\$\\$ \(Comment Specifier\)](#) token and the [* \(Comment Line Specifier\)](#) token, because these tokens cause the debugger to ignore the input text without displaying it.

.echocpnum (Show CPU Number)

Article • 10/25/2023

The **.echocpnum** command turns on or turns off the display of the processor number when you are debugging a multiprocessor target computer.

```
dbgcmd  
.echocpnum 0  
.echocpnum 1  
.echocpnum
```

Parameters

0

Turns off the display of the processor number.

1

Turns on the display of the processor number.

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about how to debug multiprocessor computers, see [Multiprocessor Syntax](#).

Remarks

If you use the **.echocpnum** command without any arguments, the display of processor numbers is turned on or off, and the new state is displayed.

By default, the display is turned off.

.echotime (Show Current Time)

Article • 10/25/2023

The **.echotime** command displays the current time.

```
dbgcmd
  .echotime
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.echotime** command displays the time to the computer that the debugger is running on.

See also

[.echotimestamps \(Show Time Stamps\)](#)

[.time \(Display System Time\)](#)

[!runaway](#)

.echotimestamps (Show Time Stamps)

Article • 10/25/2023

The `.echotimestamps` command turns on or turns off the display of time stamp information.

dbgcmd

```
.echotimestamps 0  
.echotimestamps 1  
.echotimestamps
```

Parameters

0

Turns off the display of time stamp information. This is the default behavior of the Debugger.

1

Turns on the display of time stamp information.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about `DbgPrint`, `KdPrint`, `DbgPrintEx`, and `KdPrintEx`, see "The DbgPrint Buffer" in [Reading and Filtering Debugging Messages](#).

Remarks

When you use the `.echotimestamps` command without parameters, the display of time stamps is turned on or off, and the new state is displayed.

If you turn on this display, the debugger shows time stamps for module loads, thread creations, exceptions, and other events.

The **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx** kernel-mode routines send a formatted string to a buffer on the host computer. The string is displayed in the [Debugger Command window](#) (unless you have disabled such printing). You can also display the formatted string by using the **!dbgprint** extension command.

When you use **.echotimestamps** to turn on the display of time stamps, the time and date of each comment in the **DbgPrint** buffer is displayed.

.ecxr (Display Exception Context Record)

Article • 03/07/2024

The `.ecxr` command displays the context record that is associated with the current exception.

```
dbcmd  
.ecxr
```

Environment

[] Expand table

Item	Description
Modes	User mode
Targets	Crash dump only (minidumps only)
Platforms	All

Additional Information

For more information about the register context and other context settings, see [Changing Contexts](#).

Remarks

The `.ecxr` command locates the current exception's context information and displays the important registers for the specified context record.

This command also instructs the debugger to use the context record that is associated with the current exception as the register context. After you run `.ecxr`, the debugger can access the most important registers and the stack trace for this thread. This register context persists until you enable the target to execute, change the current process or thread, or use another register context command ([.cxr](#) or `.ecxr`). For more information about register contexts, see [Register Context](#).

The `.excr` command is a synonym command and has identical functionality.

See also

[.excr](#)

[Changing Contexts](#)

[Register Context](#)

.effmach (Effective Machine)

Article • 10/25/2023

The **.effmach** command displays or changes the processor mode that the debugger uses.

dbgcmd

```
.effmach [MachineType]
```

Parameters

MachineType

Specifies the processor type that the debugger uses for this session. If you omit this parameter, the debugger displays the current machine type.

You can enter one of the following machine types.

Machine type	Description
.	Use the processor mode of the target computer's native processor mode.
#	Use the processor mode of the code that is executing for the most recent event.
x86	Use an x86-based processor mode.
amd64	Use an x64-based processor mode.
ebc	Use an EFI byte code processor mode.
arm	Use an Arm64 processor mode.
chpe	Use a CHPE processor mode.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump

Item	Description
Platforms	All

Remarks

The processor mode influences many debugger features:

- Which processor is used for stack tracing.
- Whether the process uses 32-bit or 64-bit pointers.
- Which processor's register set is active.

.enable_long_status (Enable Long Integer Display)

Article • 10/25/2023

The `.enable_long_status` command specifies whether the debugger displays long integers in decimal format or in the default radix.

dbgcmd

```
.enable_long_status 0  
.enable_long_status 1
```

Parameters

0

Displays all long integers in decimal format. This is the default behavior of the debugger.

1

Displays all long integers in the default radix.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The `.enable_long_status` command affects the output of the [dt \(Display Type\)](#) command.

In WinDbg, `.enable_long_status` also affects the display in the [Locals window](#) and the Watch window. These windows are automatically updated after you issue `.enable_long_status`.

This command affects only the display of long integers. To control whether standard integers are displayed in decimal format or the default radix, use the [.force_radix_output \(Use Radix for Integers\)](#) command.

Note The [dv \(Display Local Variables\)](#) command always displays long integers in the current number base.

To change the default radix, use the [n \(Set Number Base\)](#) command.

.enable_unicode (Enable Unicode Display)

Article • 10/25/2023

The **.enable_unicode** command specifies whether the debugger displays USHORT pointers and arrays as Unicode strings.

```
dbgcmd  
.enable_unicode 0  
.enable_unicode 1
```

Parameters

0

Displays all 16-bit (USHORT) arrays and pointers as short integers. This is the default behavior of the debugger.

1

Displays all 16-bit (USHORT) arrays and pointers as Unicode strings.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.enable_unicode** command affects the output of the [dt \(Display Type\)](#) command.

In WinDbg, the **.enable_unicode** command also affects the display in the [Locals window](#) and the Watch window. These windows are automatically updated after you issue **.enable_unicode**.

You can also select or clear **Display 16-bit values as Unicode** on the shortcut menu of the Locals or Watch window to specify the display for USHORT arrays and pointers.

See also

[ds, dS \(Display String\)](#)

.endpsrv (End Process Server)

Article • 10/25/2023

The **.endpsrv** command causes the current process server or KD connection server to close.

```
dbgcmd
```

```
.endpsrv
```

Environment

You can use this command only when you are performing remote debugging through a process server or KD connection server.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about these servers, see [Process Servers \(User Mode\)](#) or [KD Connection Servers \(Kernel Mode\)](#)

Remarks

The **.endpsrv** command terminates the process server or KD connection server currently connected to your smart client.

If you wish to terminate a process server or KD connection server from the computer on which it is running, use Task Manager to end the process (dbgsrv.exe or kdsrv.exe).

The **.endpsrv** command can terminate a process server or KD connection server, but it cannot terminate a debugging server. For information on how to do that, see [Controlling a Remote Debugging Session](#).

.endsrv (End Debugging Server)

Article • 12/05/2024

The `.endsrv` command causes the debugger to cancel an active debugging server.

```
dbgcmd  
.endsrv ServerID
```

Parameters

ServerID

Specifies the ID of the debugging server.

Environment

You can use this command only when you are performing remote debugging through the debugger.

[+] Expand table

Item	Description
Modes	User mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about remote debugging, see [Remote Debugging Through the Debugger](#).

ⓘ Important

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Remarks

You must issue the `.endsrv` command from the debugging server or from one of the debugging clients that are connected to the debugging server.

To determine the ID of a debugging server, use the [.servers \(List Debugging Servers\)](#) command.

The `.endsrv` command can terminate a debugging server, but it cannot terminate a process server or KD connection server. For information on how to end these servers, see [Controlling a Process Server Session](#) and [Controlling a KD Connection Server Session](#). (There is, however, one exceptional case when `.endsrv` can end a process server that has been launched programmatically; for details, see [IDebugClient::StartProcessServer](#).)

If you cancel a debugging server, you prevent any future debugging clients from attaching to the server. However, if you cancel a debugging server, you do not detach any clients that are currently attached through the server.

Consider the following situation. Suppose that you start some debugging servers, as the following example shows.

```
dbgcmd

0:000> .server npipe:pipe=rabbit
Server started with 'npipe:pipe=rabbit'
0:000> .server tcp:port=7
Server started with 'tcp:port=7'
```

Then, you decide to use a password, as the following example shows.

```
dbgcmd

0:000> .server npipe:pipe=tiger,password=Password
Server started with 'npipe:pipe=tiger,password=Password'
```

But the earlier servers are still running, so you should cancel them, as the following example shows.

```
dbgcmd

0:000> .servers
0 - Debugger Server - npipe:Pipe=rabbit
1 - Debugger Server - tcp:Port=7
2 - Debugger Server - npipe:Pipe=tiger,Password=*
```

```
0:000> .endsrv 0
Server told to exit. Actual exit may be delayed until
the next connection attempt.
0:000> .endsrv 1
Server told to exit. Actual exit may be delayed until
the next connection attempt.
0:000> .servers
0 - <Disabled, exit pending>
1 - <Disabled, exit pending>
2 - Debugger Server - npipe:Pipe=tiger,Password=*
```

Finally, to make sure that nothing attached to your computer while the earlier servers were active, use the [.clients \(List Debugging Clients\)](#) command.

```
dbgcmd

0:000> .clients
HotMachine\HostUser, last active Mon Mar 04 16:05:21 2002
```

Important

Using a password with TCP, NPIPE, or COM protocol offers only a small amount of protection, because the password is not encrypted. When you use a password together with a SSL or SPIPE protocol, the password is encrypted. If you want to establish a secure remote session, you must use the SSL or SPIPE protocol.

Feedback

Was this page helpful?

 Yes

 No

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

.enumtag (Enumerate Secondary Callback Data)

Article • 04/03/2024

The **.enumtag** command displays secondary bug check callback data and all data tags.

```
dbgcmd
  .enumtag
```

Environment

[+] Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information and for other ways of displaying this data, see [Reading Bug Check Callback Data](#).

Remarks

You can use the **.enumtag** command only after a bug check has occurred or when you are debugging a kernel-mode crash dump file.

For each block of secondary bug check callback data, the **.enumtag** command displays the tag (in GUID format) and the data (as bytes and ASCII characters).

Consider the following example.

```
dbgcmd
  kd> .enumtag
  {87654321-0000-0000-0000000000000000} - 0xf9c bytes
```

```
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 .....@.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....'.
.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....'.
{12345678-0000-0000-0000000000000000} - 0x298 bytes
F4 BF 7B 80 F4 BF 7B 80 00 00 00 00 00 00 00 00 ..{....{.....
4B 44 42 47 98 02 00 00 00 20 4D 80 00 00 00 00 KDBG..... M.....
54 A5 57 80 00 00 00 A0 50 5A 80 00 00 00 00 T.W.....PZ.....
40 01 08 00 18 00 00 BC 7D 50 80 00 00 00 00 @.....}P.....
.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....'.
.....
```

In this example, the first batch of secondary data has a GUID `{87654321-0000-0000-0000000000000000}` as its tag, and the second batch of data has a GUID `({12345678-0000-0000-0000000000000000})` as its tag. However, the data is not in a useful format.

Feedback

Was this page helpful?

 Yes

 No

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

.event_code (Display Event Code)

Article • 10/25/2023

The `.event_code` command displays the current event instructions.

```
dbgcmd
```

```
  .event_code
```

Environment

Item	Description
Modes	user mode, kernel mode
Targets	live debugging only
Platforms	all

Remarks

The `.event_code` command displays the hexadecimal instructions at the current event's instruction pointer. The display includes up to 64 bytes of instructions if they are available.

.eventlog (Display Recent Events)

Article • 10/25/2023

The **.eventlog** command displays the recent Microsoft Win32 debug events, such as module loading, process creation and termination, and thread creation and termination.

```
dbgcmd
```

```
.eventlog
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.eventlog** command shows only 1024 characters.

The following example shows the **.eventlog** command.

```
dbgcmd
```

```
0:000> .eventlog
0904.1014: Load module C:\Windows\system32\ADVAPI32.dll at 000007fe`fed80000
0904.1014: Load module C:\Windows\system32\RPCRT4.dll at 000007fe`fe8c0000
0904.1014: Load module C:\Windows\system32\GDI32.dll at 000007fe`fea00000
0904.1014: Load module C:\Windows\system32\USER32.dll at 00000000`76b10000
0904.1014: Load module C:\Windows\system32\msvcrt.dll at 000007fe`fe450000
0904.1014: Load module C:\Windows\system32\COMDLG32.dll at 000007fe`fecf0000
0904.1014: Load module C:\Windows\system32\SHLWAPI.dll at 000007fe`fe1f0000
0904.1014: Load module C:\Windows\WinSxS\amd64_microsoft.windows.common-
controls_6595b6414
4ccf1df_6.0.6000.16386_none_1559f1c6f365a7fa\COMCTL32.dll at
000007fe`fbda0000
0904.1014: Load module C:\Windows\system32\SHELL32.dll at 000007fe`fd4a0000
0904.1014: Load module C:\Windows\system32\WINSPOOL.DRV at 000007fe`f77d0000
0904.1014: Load module C:\Windows\system32\ole32.dll at 000007fe`feb10000
0904.1014: Load module C:\Windows\system32\OLEAUT32.dll at 000007fe`feeb0000
Last event: Break instruction exception - code 80000003 (first chance)
```

.exdicmd (EXDI Command)

Article • 10/31/2024

The **.exdicmd** sends an Extended Debugging Interface (EXDI) command to the target system using the active EXDI debugging connection. For more information about EXDI see, [Configuring the EXDI Debugger Transport](#).

dbgcmd
exdicmd component target parameters

This command passes the parameters directly through to an EXDI component. Refer to the documentation for your EXDI component for more information about the valid commands that are available for your target system.

Not all EXDI components have this function implemented.

Parameters

These are the valid parameters for **.exdicmd**.

[+] Expand table

Parameter	Description
<code>target:*:<string></code>	Pass through the <code><string></code> function to the target end entity for all processor cores.
<code>target:<n>:<string></code>	Pass through the <code><string></code> function to the target end entity for the processor core n (n-decimal number).
<code>component:*:<string></code>	Execute a EXDI component <code><string></code> function on all processor cores.
<code>component:<n>:<string></code>	Execute a EXDI component <code><string></code> function on the processor core n (n-decimal number).
<code>help</code>	Display basic help.

Target exdicmd usage

`.exdicmd target:*:<string>`

Using the target parameter provides a way to communicate between Windows debugger and the EXDI COM server. The debugger will display the command result if the command returns a response back to the debugger engine.

The .exdicmd target parameter will take any command that EXDI COM server is able to process. This allows for command usage beyond what is supported directly in the EXDI interface.

Component exdicmd usage

```
.exdicmd <component>:
```

The purpose for this command is primarily to be able to execute/test EXDI COM server functions to validate basic functionality. It is typically used less than the target parameter usage described above.

Note that there are commands that can be acted on internally by the EXDI COM server without it needing to send it on to the JTAG target entity. For example, it is possible to collect telemetry from the EXDI COM server to validate its correct operation.

Environment

[+] Expand table

Descriptor	Value
Modes	Kernel mode only
Target	Live debugging only
Platforms	All

Additional Information

Example use of .exdicmd with an OpenOCD target, which uses the syntax, `.exdicmd target:0:<OpenOCD command>` is shown below.

```
dbgcmd

0: kd> .exdicmd target:0:info network
Target command response: e1000.0:
index=0,type=nic,model=e1000,macaddr=52:54:00:12:34:56
 \ net0: index=0,type=user,net=10.0.2.0,restrict=off
```

```
OK
exdiCmd: The function: 'info network' was completed.
```

```
dbgcmd
```

```
0: kd> .exdicmd target:0:info registers system -v
Target command response:
NumberOfRegisters: 9
```

Name	Value	Access code
fs_base	0000000000000000	n/a
gs_base	fffff8047b907000	n/a
k_gs_base	000000e7cbdbe000	n/a
cr0	000000080050033	n/a
cr2	fffff8048454de64	n/a
cr3	00000000001ae000	n/a
cr4	0000000000006f8	n/a
cr8	0000000000000000	n/a
efer	000000000000d01	0xc0000080

```
exdiCmd: The function: 'info registers system -v' was completed.
```

If the target system is not able to understand the command, and *unknown command* message will be returned.

```
dbgcmd
```

```
0: kd> .exdicmd target:0:Foo
Target command response: unknown command: 'Foo'
```

.outmask

If output is not displayed check that the output mask is set to display verbose output using [.outmask \(Control Output Mask\)](#).

If the debugger was not launched in verbose mode, the .outmask can be set using `.outmask 7FF`.

Remarks

For more information on setting up an EXDI debugger connection, see [Configuring the EXDI Debugger Transport](#).

Feedback

Was this page helpful?

 Yes

 No

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

.exepath (Set Executable Path)

Article • 10/25/2023

The **.exepath** command sets or displays the executable file search path.

dbgcmd

```
.exepath[+] [Directory [; ...]]
```

Parameters

+

Specifies that the debugger should append the new directories to the previous executable file search path (instead of replacing the path).

Directory

Specifies one or more directories to put in the search path. If you do not specify *Directory*, the current path is displayed. You can separate multiple directories with semicolons.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

In most situations, the debugger knows the location of the executable files, so you do not have to set the path for this file.

However, there are situations when this path is required. For example, kernel-mode small memory dump files do not contain all of the executable files that exist in memory at the time of a stop error (that is, a crash). Similarly, user-mode minidump files do not contain the application binaries. If you set the path of the executable files, the debugger

can find these binary files. For more information, see [Setting Symbol and Executable Image Paths in WinDbg](#).

The executable file search path can also be set using the `_NT_EXECUTABLE_IMAGE_PATH` environment variable. For more information, see [General Environment Variables](#).

See also

[Setting Symbol and Executable Image Paths in WinDbg](#)

[General Environment Variables](#)

.expr (Choose Expression Evaluator)

Article • 10/25/2023

The **.expr** command specifies the default expression evaluator.

```
dbgcmd  
.expr /s masm  
.expr /s c++  
.expr /q  
.expr
```

Parameters

/s masm

Changes the default expression type to Microsoft Assembler expression evaluator (MASM). This type is the default value when you start the debugger.

/s c++

Changes the default expression type to the C++ expression evaluator.

/q

Displays the list of possible expression types.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When you use the **.expr** command without an argument, the debugger displays the current default expression type.

The [?? \(Evaluate C++ Expression\)](#) command, the Watch window, and the [Locals window](#) always use C++ expression syntax. All other commands and debugging information windows use the default expression evaluator.

For more information about how to control which syntax is used, see [Evaluating Expressions](#). For more information about the syntax, see [Numerical Expression Syntax](#).

See also

[? \(Evaluate Expression\)](#)

[MASM Numbers and Operators](#)

[C++ Numbers and Operators](#)

.exptr (Display Exception Pointers)

Article • 10/25/2023

The `.exptr` command displays an EXCEPTION_POINTERS structure.

```
dbgcmd
```

```
  .exptr Address
```

Parameters

Address

Specifies the address of the EXCEPTION_POINTERS structure.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.exr (Display Exception Record)

Article • 03/07/2024

The `.exr` command displays the contents of an exception record.

```
dbgcmd  
.exr Address  
.exr -1
```

Parameters

Address - Specifies the address of the exception record. If you specify `-1` as the address, the debugger displays the most recent exception.

Environment

[] [Expand table](#)

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The `.exr` command displays information that is related to an exception that the debugger encountered on the target computer. The information that is displayed includes the exception address, the exception code, the exception flags, and a variable list of parameters to the exception.

You can usually obtain the *Address* by using the [!pcr](#) extension.

The `.exr` command is often used to debug bug check 0x1E. For more information and an example, see [Bug Check 0x1E](#) (KMODE_EXCEPTION_NOT_HANDLED).

See also

Changing Contexts

Register Context

.ecxr (Display Exception Context Record)

.trap

.excr (Display Exception Context Record)

Article • 03/07/2024

The `.excr` command displays the context record that is associated with the current exception.

```
dbgcmd
  .excr
```

For more information about the register context and other context settings, see [Changing Contexts](#).

Environment

[+] Expand table

Item	Description
Modes	User mode only
Targets	Crash dump only (minidumps only)
Platforms	All

Remarks

The `.excr` command locates the current exception's context information and displays the important registers for the specified context record.

This command also instructs the debugger to use the context record that is associated with the current exception as the register context. After you run `.excr`, the debugger can access the most important registers and the stack trace for this thread. This register context persists until you enable the target to execute, change the current process or thread, or use another register context command ([.cxr](#) or [.excr](#)). For more information about register contexts, see [Register Context](#).

The [.ecxr](#) command is a synonym command and has identical functionality.

See also

Changing Contexts

Register Context

.ecxr

.extmatch (Display All Matching Extensions)

Article • 10/25/2023

The **.extmatch** command displays extension commands exported by the currently loaded extension DLLs that match the specified pattern.

```
dbgcmd  
.extmatch [Options] Pattern
```

Parameters

Options Specifies the searching options. You can use one or more of the following options:

/e ** ExtDLLPattern**

Limits the enumeration to extension commands exported by extension DLLs that match the specified pattern string. *ExtDLLPattern* is matched against the extension DLL filename.

/n

Excludes the extension name when the extensions are displayed. Thus, if this option is specified, then only the extension name itself will be displayed.

/D

Displays the output using [Debugger Markup Language \(DML\)](#). In the output, each listed extension is a link that you can click to get more information about the extension. Not all extension modules support DML.

Pattern

Specifies a pattern that the extension must contain. *Pattern* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode

Item	Description
Targets	Live, crash dump
Platforms	All

Remarks

To display a list of loaded extension DLLs, use the [.chain](#) command.

Here is an example of this command, showing all the loaded extension DLLs that have an export named !help:

```
dbgcmd

0:000> .extmatch help
!ext.help
!exts.help
!uext.help
!ntsdexts.help
```

The following example lists all extension commands beginning with the string "he" that are exported by extension DLLs whose names begin with the string "ex":

```
dbgcmd

0:000> .extmatch /e ext* he*
!ext.heap
!ext.help
!exts.heap
!exts.help
```

The following example lists all extension commands, so we can see which ones support DML.



.extpath (Set Extension Path)

Article • 10/25/2023

The `.extpath` command sets or displays the extension DLL search path.

dbgcmd

```
.extpath[+] [Directory[;...]]
```

Parameters

`+`

Signifies that the debugger should append new directories to the previous extension DLL search path (instead of replacing the path).

Directory

Specifies one or more directories to put in the search path. If you do not specify *Directory*, the current path is displayed. You can separate multiple directories with semicolons.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the extension search path and loading extension DLLs, see [Loading Debugger Extension DLLs](#).

Remarks

The extension DLL search path is reset to its default value at the start of each debugging session.

During live kernel-mode debugging, a reboot of the target computer results in the start of a new debugging session. So any changes that you make to the extension DLL search path during kernel-mode debugging will not persist across a reboot of the target computer.

The default value of the extension DLL search path contains all the extension paths known to the debugger and all the paths in the %PATH% environment variable. For example, suppose your %PATH% environment variable has a value of `C:\Windows\system32;C:\Windows`. Then the default value of the DLL extension search path might look like this.

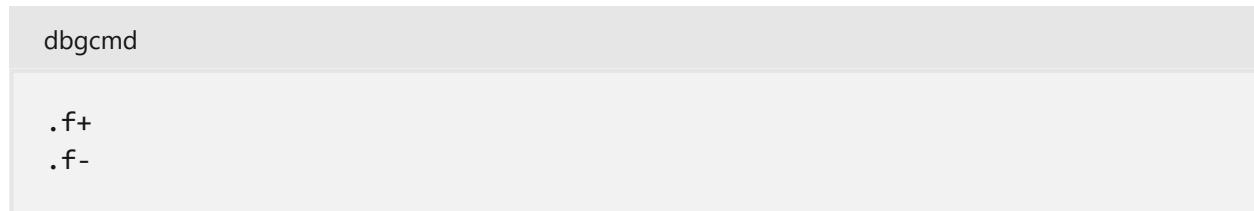
```
dbgcmd

0:000> .extpath
Extension search path is: C:\Program Files\Debugging Tools for Windows
(x64)\WINXP;C:\Program Files\
Debugging Tools for Windows (x64)\winext;C:\Program Files\Debugging Tools
for Windows (x64)\winext\
arcade;C:\Program Files\Debugging Tools for Windows (x64);C:\Program
Files\Debugging Tools for
Windows (x64)\winext\arcade;C:\Windows\system32;C:\Windows
```

.f+, .f- (Shift Local Context)

Article • 10/25/2023

The `.f+` command shifts the frame index to the next frame in the current stack. The `.f-` command shifts the frame index to the previous frame in the current stack.



Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the local context and other context settings, see [Changing Contexts](#). For more information about how to display local variables and other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The *frame* specifies the local context (scope) that the debugger uses to interpret local variables

The `.f+` and `.f-` commands are shortcuts for moving to the next and previous frames in the current stack. These commands are equivalent to the following `.frame` commands, but the `.f` commands are shorter for convenience:

- `.f+` is the same as `.frame @$frame + 1`.
- `.f-` is the same as `.frame @$frame - 1`.

The dollar sign (\$) identifies the frame value as a [pseudo-register](#). The at sign (@) causes the debugger to access the value more quickly, because it notifies the debugger that a

string is a register or pseudo-register.

When an application is running, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function that they are defined in. Unless you use an `.f+` or `.f-` command (or a `.frame` command), the debugger uses the scope of the current function (the current frame on the stack) as the local context.

The *frame number* is the position of the stack frame within the stack trace. You can view this stack trace by using the [k, kb, kc, kd, kp, kP, kv \(Display Stack Backtrace\)](#) command or the [Calls window](#). The first line (the current frame) represents frame number 0. The subsequent lines represent frame numbers 1, 2, 3, and so on.

You can set the local context to a different stack frame to view new local variable information. However, the actual variables that are available depend on the code that is executed.

The debugger resets the local context to the scope of the program counter if any program execution occurs. The local context is reset to the top stack frame if the register context is changed.

.fiber (Set Fiber Context)

Article • 10/25/2023

The `.fiber` command specifies which fiber is used for the fiber context.

```
dbgcmd
```

```
  .fiber [Address]
```

Parameters

Address

Specifies the address of the fiber. If you omit this parameter or specify zero, the fiber context is reset to the current fiber.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the process context, the register context, and the local context, see [Changing Contexts](#).

.fiximports (Fix Target Module Imports)

Article • 10/25/2023

The `.fiximports` command validates and corrects all static import links for a target module.

```
dbgcmd
```

```
  .fiximports Module
```

Parameters

Module

Specifies the target module whose imports the debugger corrects. *Module* can contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#). If you want to include spaces in *Module*, you must enclose the parameter in quotation marks.

Environment

Modes	User mode, kernel mode
Targets	Crash dump only (minidump only)
Platforms	All

Remarks

You can use the `.fiximports` command only when the target is a minidump that does not contain its own executable images.

When the debugger maps images for use as memory, the debugger does not automatically connect image imports to exporters. Therefore, instructions that refer to imports are disassembled in the same manner as in a live debugging session. You can use `.fiximports` to request that the debugger perform the appropriate import linking.

.flash_on_break (Flash on Break)

Article • 10/25/2023

The `.flash_on_break` command specifies whether the WinDbg taskbar entry flashes when WinDbg is minimized and the target breaks.

dbgcmd

```
.flash_on_break on  
.flash_on_break off  
.flash_on_break
```

Parameters

on

Causes the WinDbg taskbar entry to flash if WinDbg is minimized and the target breaks into the debugger. This is the default behavior for WinDbg.

off

Prevents the WinDbg taskbar entry from flashing.

Environment

The `.flash_on_break` command is available only in WinDbg. You cannot use this command in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you use the `.flash_on_break` command without parameters, the debugger displays the current flash setting.

.fnent (Display Function Data)

Article • 10/25/2023

The **.fnent** command displays information about the function table entry for a specified function.

```
dbgcmd
```

```
  .fnent Address
```

Parameters

Address

Specifies the address of the function.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The symbol search algorithm for the **.fnent** command is the same as that of the [In \(List Nearest Symbols\)](#) command. The display first shows the nearest symbols. Then, the debugger displays the function entry for the first of these symbols.

If the nearest symbol is not in the function table, no information is displayed.

The following example shows a possible display.

```
dbgcmd
```

```
0:001> .fnent 77f9f9e7
Debugger function entry 00b61f50 for:
(77f9f9e7)  ntdll!RtlpBreakWithStatusInstruction  |  (77f9fa98)
ntdll!DbgPrintReturnControlC
```

```
Params:    1
```

```
Locals:    0
Registers: 0

0:001> .fnent 77f9fa98
Debugger function entry 00b61f70 for:
(77f9fa98)  ntdll!DbgPrintReturnControlC | (77f9fb21)  ntdll!DbgPrompt

Non-FPO

0:001> .fnent 01005a60
No function entry for 01005a60
```

.fnret (Display Function Return Value)

Article • 10/25/2023

The **.fnret** command displays information about a function's return value.

dbgcmd

```
.fnret [/s] Address [Value]
```

Parameters

/s

Sets the **\$callret** pseudo-register equal to the return value that is being displayed, including type information.

Address

Specifies the address of the function.

Value

Specifies the return value to display. If you include *Value*, **.fnret** casts *Value* to the return type of the specified function and displays it in the format of the return type. If you omit *Value*, the debugger obtains the return value of the function from the return value registers.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you include the *Value* parameter, the **.fnret** command only casts this value to the proper type and displays the results.

If you omit *Value*, the debugger uses the return value registers to determine this value. If a function has returned more recently than the function that the *Address* parameter

specifies, the value that is displayed will probably not be a value that this function returned.

.force_radix_output (Use Radix for Integers)

Article • 10/25/2023

The **.force_radix_output** command specifies whether integers are displayed in decimal format or in the default radix.

dbgcmd

```
.force_radix_output 0  
.force_radix_output 1
```

Parameters

0

Displays all integers (except for long integers) in decimal format. This is the default behavior for the Debugger.

1

Displays all integers (except for long integers) in the default radix.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.force_radix_output** command affects the output of the [dt \(Display Type\)](#) command.

In WinDbg, **.force_radix_output** also affects the display in the [Locals window](#) and the Watch window. You can select or clear **Always display numbers in default radix** on the shortcut menu of the Locals or Watch window to have the same effect as

`.force_radix_output`. These windows are automatically updated after you issue this command.

The `.force_radix_output` command affects only the display of standard integers. To specify whether long integers are displayed in decimal format or the default radix, use the [`.enable_long_status \(Enable Long Integer Display\)`](#) command.

To change the default radix, use the [`n \(Set Number Base\)`](#) command.

.force_tb (Forcibly Allow Branch Tracing)

Article • 10/25/2023

The `.force_tb` command forces the processor to trace branches early in the boot process.

```
dbgcmd
```

```
.force_tb
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Typically, branch tracing is enabled after the debugger initializes the processor control block (PRCB). This initialization occurs early in the boot process.

However, if you have to use the [tb \(Trace to Next Branch\)](#) command before this initialization, you can use the `.force_tb` command to enable branch tracing earlier. Use this command carefully because it can corrupt your processor state.

.formats (Show Number Formats)

Article • 10/25/2023

The **.formats** command evaluates an expression or symbol in the context of the current thread and process and displays it in multiple numeric formats.

```
dbgcmd  
.formats expression
```

Parameters

expression

Specifies the expression to evaluate. For more information about the syntax, see [Numerical Expression Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The evaluated expression is displayed in hexadecimal, decimal, octal, and binary formats, as well as single-precision and double-precision floating-point format. ASCII character formats are also displayed when the bytes correspond to standard ASCII characters. The expression is also interpreted as a time stamp if it is in the allowed range.

The following example shows a **.formats** command.

```
dbgcmd  
0:000> .formats 1c407e62  
Evaluate expression:  
Hex:      1c407e62  
Decimal: 473988706  
Octal:   03420077142
```

```
Binary: 00011100 01000000 01111110 01100010
Chars: .@~b
Time: Mon Jan 07 15:31:46 1985
Float: low 6.36908e-022 high 0
Double: 2.34182e-315
```

The **Time** field displays a 32-bit value in CRT time stamp format and displays a 64-bit value in FILETIME format. You can distinguish these formats because the FILETIME format includes milliseconds and the CRT format does not.

See also

[? \(Evaluate Expression\)](#)

[?? \(Evaluate C++ Expression\)](#)

.fpo (Control FPO Overrides)

Article • 10/25/2023

The **.fpo** command controls the frame pointer omission (FPO) overrides.

```
dbgcmd  
.fpo -s [-fFlag] Address  
.fpo -d Address  
.fpo -x Address  
.fpo -o Address  
.fpo Address
```

Parameters

-s

Sets an FPO override at the specified address.

-fFlag

Specifies FPO flags for the override. You must not add a space between **-f** and *Flag*. If the flag takes an argument, you must add a space between the flag and that argument. If you want multiple flags, you must repeat the **-f** switch (for example, **-fb -fp 4 -fe**). You can use the **-f** switch only with **-s**. You can use one of the following values for *Flag*.

Flag	Effect
b	Sets fUseBP equal to TRUE .
e	Sets fUseSEH equal to TRUE .
n	Sets cbFrame equal to FRAME_NONFPO . (By default, cbFrame is set to FRAME_FPO .)
I Term	Sets cdwLocals equal to <i>Term</i> . <i>Term</i> should specify the local DWORD count that you want.
p Term	Sets cdwParams equal to <i>Term</i> . <i>Term</i> should specify the parameter DWORD count that you want.
r Term	Sets cbRegs equal to <i>Term</i> . <i>Term</i> should specify the register count that you want.
s Term	Sets cbProcSize equal to <i>Term</i> . <i>Term</i> should specify the procedure size that you want.

Flag	Effect
t <i>Term</i>	Sets cbFrame equal to <i>Term</i> . <i>Term</i> should specify one of the following frame types: <ul style="list-style-type: none"> • FRAME_FPO 0 • FRAME_TRAP 1 • FRAME_TSS 2 • FRAME_NONFPO 3

Address

Specifies the address where the debugger sets or removes the override or the address whose overrides the debugger should display. This address must be within a module in the current module list.

-d

Removes the FPO overrides at the specified address.

-x

Removes all FPO overrides within the module that contains the *Address* address.

-o

Displays all FPO overrides within the module that contains the *Address* address.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Without parameters, the **.fpo** command displays the FPO overrides for the specified address.

Some compilers (including Microsoft Visual Studio 6.0 and earlier versions) generate FPO information to indicate how the stack frame is set up. During stack walking, the debugger uses these FPO records to understand the stack. If the compiler generated incorrect FPO information, you can use the **.fpo** command to fix this problem.

.frame (Set Local Context)

Article • 10/25/2023

The **.frame** command specifies which local context (scope) is used to interpret local variables or displays the current local context.

dbgcmd

```
.frame [/c] [/r] [FrameNumber]
.frame [/c] [/r] = BasePtr [FrameIncrement]
.frame [/c] [/r] = BasePtr StackPtr InstructionPtr
```

Parameters

/c

Sets the specified frame as the current local override context. This action allows a user to access the nonvolatile registers for any function in the call stack.

/r

Displays registers and other information about the specified local context.

FrameNumber

Specifies the number of the frame whose local context you want. If this parameter is zero, the command specifies the current frame. If you omit this parameter, this command displays the current local context.

BasePtr

Specifies the base pointer for the stack trace that is used to determine the frame, if you add an equal sign (=) after the command name (**.frame**). On an x86-based processor, you add another argument after *BasePtr* (which is interpreted as *FrameIncrement*) or two more arguments after *BasePtr* (which are interpreted as *InstructionPtr* and *StackPtr*).

FrameIncrement

(x86-based processor only)

Specifies an additional quantity of frames past the base pointer. For example, if the base pointer 0x0012FF00 is the address of frame 3, the command **.frame 12ff00** is equivalent to **.frame 3**, and **.frame 12ff00 2** is equivalent to **.frame 5**.

StackPtr

(x86-based processor only) Specifies the stack pointer for the stack trace that is used to determine the frame. If you omit *StackPtr* and *InstructionPtr*, the debugger uses the

stack pointer that the **esp** register specifies and the instruction pointer that the **eip** register specifies.

InstructionPtr

(x86-based processor only) Specifies the instruction pointer for the stack trace that is used to determine the frame. If you omit *StackPtr* and *InstructionPtr*, the debugger uses the stack pointer that the **esp** register specifies and the instruction pointer that the **eip** register specifies.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the local context and other context settings, see [Changing Contexts](#). For more information about how to display local variables and other memory-related commands, see [Reading and Writing Memory](#).

Remarks

When an application is running, the meaning of local variables depends on the location of the program counter, because the scope of such variables extends only to the function that they are defined in. If you do not use the **.frame** command, the debugger uses the scope of the current function (the current frame on the stack) as the [local context](#).

To change the local context, use the **.frame** command and specify the frame number that you want.

The *frame number* is the position of the stack frame within the stack trace. You can view this stack trace with the [k \(Display Stack Backtrace\)](#) command or the [Calls window](#). The first line (the current frame) is frame number 0. The subsequent lines represent frame numbers 1, 2, 3, and so on.

If you use the **n** parameter with the **k** command, the **k** command displays frame numbers together with the stack trace. These frame numbers are always displayed in

hexadecimal form. On the other hand, the `.frame` command interprets its argument in the default radix, unless you override this setting with a prefix such as `0x`. To change the default radix, use the [n \(Set Number Base\)](#) command.

You can set the local context to a different stack frame to enable you to view new local variable information. However, the actual variables that are available depend on the code that is being executed.

The local context is reset to the scope of the program counter if any application execution occurs. The local context is reset to the top stack frame if the register context is changed.

.help (Meta-Command Help)

Article • 10/25/2023

The `.help` command displays a list of all meta-commands.

```
dbgcmd
```

```
.help  
.help /D
```

Parameters

`/D`

Displays output using [Debugger Markup Language](#). The output contains a list of links that you can click to see the meta-commands that begin with a particular letter.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

For more information about meta-commands, use the `.help` command. For more information about standard commands, use the `?` command. For more information about extension commands, use the `!help` extension.

.hh (Open HTML Help File)

Article • 10/25/2023

The **.hh** command opens the Debugging Tools for Windows documentation.

dbgcmd
.hh [Text]

Parameters

Text

Specifies the text to find in the index of the Help documentation.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

You cannot use this command when you are performing [remote debugging through Remote.exe](#).

Additional Information

For more information about the Help documentation, see [Using the Help File](#).

Remarks

The **.hh** command opens the Debugging Tools for Windows documentation (Debugger.chm). If you specify *Text*, the debugger opens the **Index** pane in the documentation and searches for *Text* as a keyword in the index. If you do not specify *Text*, the debugger opens the **Contents** pane of the documentation.

.hideinjectedcode (Hide Injected Code)

Article • 10/25/2023

The **.hideinjectedcode** command turns on, or off, the hiding of injected code.

dbgcmd

```
.hideinjectedcode { on | off }
.hideinjectedcode help
.hideinjectedcode
```

Parameters

on

Turns on the hiding of injected code.

off ****

Turns off the hiding of injected code

help

Displays help for this command.

Remarks

If you enter this command with no parameter, it displays help for the command.

.holdmem (Hold and Compare Memory)

Article • 10/25/2023

The **.holdmem** command saves memory ranges and compares them to other memory ranges.

```
dbgcmd

.holdmem -a Range
.holdmem -d { Range | Address }
.holdmem -D
.holdmem -o
.holdmem -c Range
```

Parameters

-a ** *Range***

Specifies the memory range to save. For more information about the syntax, see [Address and Address Range Syntax](#).

-d ** { *Range* | *Address* }**

Specifies memory ranges to delete. If you specify *Address*, the debugger deletes any saved range that contains that address. If you specify *Range*, the debugger deletes any saved range that overlaps with *Range*. For more information about the syntax, see [Address and Address Range Syntax](#).

-D

Deletes all saved memory ranges.

-o

Displays all saved memory ranges.

-c *Range*

Compares the specified range to all saved memory ranges. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode

Item	Description
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to manipulate memory and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

The **.holdmem** command compares memory ranges byte-for-byte.

If any of the specified memory locations do not exist in the virtual address space, the command returns an error.

.idle_cmd (Set Idle Command)

Article • 10/25/2023

The `.idle_cmd` command sets the *idle command*. This is a command that is executed whenever control returns from the target to the debugger. For example, when the target reaches a breakpoint, this command executes.

dbgcmd

```
.idle_cmd  
.idle_cmd String  
.idle_cmd /d
```

Parameters

String

Specifies the string to which the idle command should be set.

`/d`

Clears the idle command.

Environment

This command cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When `.idle_cmd` is used with no parameters it displays the current idle command.

In WinDbg, idle commands are stored in workspaces.

Here is an example. The idle command is set to `r eax`. Then, because the debugger is already idle, this command immediately executes, displaying the `eax` register:

dbgcmd

```
windbg> .idle_cmd r eax
Execute when idle: r eax
eax=003b0de8
```

.ignore_missing_pages (Suppress Missing Page Errors)

Article • 05/20/2024

The `.ignore_missing_pages` command suppresses the error messages when a Kernel Memory Dump has missing pages.

```
dbgcmd  
.ignore_missing_pages 0  
.ignore_missing_pages 1  
.ignore_missing_pages
```

Parameters

0

Displays all missing page errors while debugging a Kernel Memory Dump. This is the default behavior of the debugger.

1

Suppresses the display of missing page errors while debugging a Kernel Memory Dump.

Environment

[] Expand table

Modes	Kernel mode only
Targets	Dump file debugging only
Platforms	All

Additional Information

For more information about how to debug these dump files, see [Kernel Memory Dump](#).

Remarks

Without parameters, `.ignore_missing_pages` displays the current status of this setting.

Feedback

Was this page helpful?

 Yes

 No

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

.inline (Toggle Inline Function Debugging)

Article • 10/25/2023

The `.inline` command enables or disables inline function debugging.

dbgcmd

```
.inline 0  
.inline 1
```

Parameters

0

Disables inline function debugging.

1

Enables inline function debugging.

See also

[Debugging Optimized Code and Inline Functions](#)

.imgscan (Find Image Headers)

Article • 10/25/2023

The **.imgscan** command scans virtual memory for image headers.

```
dbgcmd  
.imgscan [Options]
```

Parameters

Options Any of the following options:

/r ** Range**

Specifies the range to search. For more information about the syntax, see [Address and Address Range Syntax](#). If you specify only one address, the debugger searches a range that begins at that address and extends 0x10000 bytes.

/l

Loads module information for any image header that is found.

/v

Displays verbose information.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you do not use the **/r** parameter, the debugger searches all virtual memory regions.

The **.imgscan** command displays any image headers that it finds and the header type. Header types include Portable Executable (PE) headers and Microsoft MS-DOS MZ headers.

The following example shows the .imgscan command.

```
dbgcmd
```

```
0:000> .imgscan
MZ at 00400000, prot 00000002, type 01000000 - size 2d000
MZ at 77f80000, prot 00000002, type 01000000 - size 7d000
  Name: ntdll.dll
MZ at 7c570000, prot 00000002, type 01000000 - size b8000
  Name: KERNEL32.dll
```

.jdinfo (Use JIT_DEBUG_INFO)

Article • 10/25/2023

The `.jdinfo` command uses a `JIT_DEBUG_INFO` structure as the source of the exception and context for just in time (JIT) debugging. The address to the structure is passed to the `.jdinfo` command using the `%p` parameter that is specified in the `AeDebug` registry entry.

For more information about the registry keys used, see [Enabling Postmortem Debugging](#). For more information about register contexts, see [Changing Contexts](#).

```
dbgcmd
```

```
.jdinfo Address
```

Parameters

Address

Specifies the address of the `JIT_DEBUG_INFO` structure. The address to the structure is passed to the `.jdinfo` command using the `%p` parameter that is specified in the `AeDebug` registry entry.

Environment

Modes	User mode
Targets	Live, crash dump
Platforms	All

Example

This example show how the `AeDebug` registry entry can be configured to use the `WinDbg` can be used as the JIT debugger.

```
dbgcmd
```

```
Debugger = "Path\WinDbg.EXE -p %ld -e %ld -c ".jdinfo 0x%p"
```

Then, when a crash occurs, the configured JIT debugger is invoked and the %p parameter is used to pass the address of the JIT_DEBUG_INFO structure to the .jdinfo command that is executed after the debugger is started.

```
dbgcmd

nMicrosoft (R) Windows Debugger Version 10.0.10240.9 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Executable search path is:
...
ModLoad: 00000000`68a20000 00000000`68ac3000
C:\WINDOWS\WinSxS\amd64_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.9247_n
one_08e394a1a83e212f\MSVCR90.dll
(153c.5d0): Break instruction exception - code 80000003 (first chance)
Processing initial command '.jdinfo 0x00000000003E0000'
ntdll!DbgBreakPoint:
00007ffc`81a986a0 cc           int      3
0:003> .jdinfo 0x00000000003E0000
----- Exception occurred on thread 0:15c8
ntdll!ZwWaitForMultipleObjects+0x14:
00007ffc`81a959a4 c3           ret

----- Exception record at 00000000`003e0028:
ExceptionAddress: 00007ff791d81014 (CrashAV_x64!wmain+0x0000000000000014)
    ExceptionCode: c0000005 (Access violation)
    ExceptionFlags: 00000000
NumberParameters: 2
    Parameter[0]: 0000000000000001
    Parameter[1]: 0000000000000000
Attempt to write to address 0000000000000000

----- Context record at 00000000`003e00c0:
rax=0000000000000000 rbx=0000000000000000 rcx=00007ffc81a954d4
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000001
rip=00007ff791d81014 rsp=00000000006ff8b0 rbp=0000000000000000
    r8=00000000006ff808 r9=0000000000000000 r10=0000000000000000
    r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
    r14=0000000000000000 r15=0000000000000000
    iopl=0          nv up ei pl zr na po nc
    cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b
    efl=00010246
CrashAV_x64!wmain+0x14:
00007ff7`91d81014 45891b         mov     dword ptr [r11],r11d
ds:00000000`00000000=????????
```

Remarks

The **.jdinfo** command uses the **AeDebug** registry information introduced in Windows Vista. For more information about the registry keys used, see [Enabling Postmortem Debugging](#). The **.jdinfo** command takes the address of a **JIT_DEBUG_INFO** that the system set up for **AeDebug** and sets the context to the exception that caused the crash.

You can use the **.jdinfo** command instead of **-g** in **AeDebug** to have your debugger set to the **AeDebug** state without requiring execution.

This state can be advantageous, because under usual conditions, when a user-mode exception occurs, the following sequence occurs:

1. The Microsoft Windows operating system halts execution of the application.
2. The postmortem debugger is started.
3. The debugger attaches to the application.
4. The debugger issues a "Go" command. (This command is caused by the **-g** in the **AeDebug** key.)
5. The target attempts to execute and may or may not encounter the same exception.
6. This exception breaks into the debugger.

There are several problems that can occur because of these events:

- Exceptions do not always repeat, possibly because of a transient condition that no longer exists when the exception is restarted.
- Another event, such as a different exception, might occur. There is no way of knowing whether it is identical to the original event.
- Attaching a debugger involves injecting a new thread, which can be blocked if a thread is holding the loader lock. Injecting a new thread can be a significant disturbance of the process.

If you use **-c .jdinfo** instead of **-g** in your **AeDebug** key, no execution occurs. Instead, the exception information is retrieved from the **JIT_DEBUG_INFO** structure using the **%p** variable.

For example, consider the following **AeDebug** key.

```
dbgcmd
```

```
ntsd -p %ld -e %ld -c ".jdinfo 0x%p"
```

The following example is even less invasive. The **-pv** switch causes the debugger to attach noninvasively, which does not inject any new threads into the target.

```
dbgcmd  
ntsd -pv -p %ld -e %ld -c ".jdinfo 0x%p"
```

If you use this noninvasive option, exiting the debugger does not end the process. You can use the [.kill \(Kill Process\)](#) command to end the process.

If you want to use this for dump file debugging, you should use [.dump /j](#) to add the `JIT_DEBUG_INFO` structure to your dump file, when the dump file is created.

The `JIT_DEBUG_INFO` structure is defined as follows.

```
dbgcmd  
  
typedef struct _JIT_DEBUG_INFO {  
    DWORD dwSize;  
    DWORD dwProcessorArchitecture;  
    DWORD dwThreadID;  
    DWORD dwReserved0;  
    ULONG64 lpExceptionAddress;  
    ULONG64 lpExceptionRecord;  
    ULONG64 lpContextRecord;  
} JIT_DEBUG_INFO, *LPJIT_DEBUG_INFO;
```

You can use the `dt` command to display the `JIT_DEBUG_INFO` structure.

```
dbgcmd  
  
0: kd> dt JIT_DEBUG_INFO  
nt!JIT_DEBUG_INFO  
+0x000 dwSize          : Uint4B  
+0x004 dwProcessorArchitecture : Uint4B  
+0x008 dwThreadID      : Uint4B  
+0x00c dwReserved0     : Uint4B  
+0x010 lpExceptionAddress : Uint8B  
+0x018 lpExceptionRecord : Uint8B  
+0x020 lpContextRecord  : Uint8B
```

Viewing the Exception Record, Call Stack and LastEvent Using WinDbg

After the `jdinfo` command has been used to set the context to the moment of failure, you can view the exception record returned by `jdinfo`, the call stack and the `lastevent`, as shown below, to investigate cause.

dbgcmd

```
0:000> .jdinfo 0x00000000003E0000
----- Exception occurred on thread 0:15c8
ntdll!NtWaitForMultipleObjects+0x14:
00007ffc`81a959a4 c3          ret

----- Exception record at 00000000`003e0028:
ExceptionAddress: 00007ff791d81014 (CrashAV_x64!wmain+0x0000000000000014)
  ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
  Parameter[0]: 0000000000000001
  Parameter[1]: 0000000000000000
Attempt to write to address 0000000000000000
...
0:000> k
*** Stack trace for last set context - .thread/.cxr resets it
# Child-SP      RetAddr      Call Site
00 00000000`006ff8b0 00007ff7`91d811d2 CrashAV_x64!wmain+0x14
[c:\my\my_projects\crash\crashav\crashav.cpp @ 14]
01 00000000`006ff8e0 00007ffc`7fa38364 CrashAV_x64!_tmainCRTStartup+0x11a
[f:\dd\vctools\crt_bld\self_64_amd64\crt\src\crtexe.c @ 579]
02 00000000`006ff910 00007ffc`81a55e91 KERNEL32!BaseThreadInitThunk+0x14
03 00000000`006ff940 00000000`00000000 ntdll!RtlUserThreadStart+0x21

0:000> .lastevent
Last event: 153c.5d0: Break instruction exception - code 80000003 (first
chance)
debugger time: Thu Sep  8 12:55:08.968 2016 (UTC - 7:00)
```

.kdfiles (Set Driver Replacement Map)

Article • 10/25/2023

The `.kdfiles` command reads a file and uses its contents as the driver replacement map.

```
dbgcmd  
  
.kdfiles MapFile  
.kdfiles -m OldDriver NewDriver  
.kdfiles -s SaveFile  
.kdfiles -c  
.kdfiles
```

Parameters

MapFile

Specifies the driver replacement map file to read.

-m

Adds a driver replacement association to the current association list.

OldDriver

Specifies the path and file name of the previous driver on the target computer. The syntax for *OldDriver* is the same as that of the first line after `map` in a driver replacement file. For more information about this syntax, see [Mapping Driver Files](#).

NewDriver

Specifies the path and file name of the new driver. This driver can be on the host computer or at some other network location. The syntax for *NewDriver* is the same as that of the second line after `map` in a driver replacement file. For more information about this syntax, see [Mapping Driver Files](#).

-s

Creates a file and writes the current driver replacement associations to that file.

SaveFile

Specifies the name of the file to create.

-c

Deletes the existing driver replacement map. (This option does not alter the map file itself. Instead, this option clears the debugger's current map settings.)

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	x86-based processors

Additional Information

For more information about and examples of driver replacement and the replacement of other kernel-mode modules, a description of the format for driver replacement map files, and restrictions for using this feature, see [Mapping Driver Files](#).

Remarks

If you use the `.kdfiles` command without parameters, the debugger displays the path and name of the current driver replacement map file and the current set of replacement associations.

When you run this command, the specified *MapFilefile* is read. If the file is not found or if it does not contain text in the proper format, the debugger displays a message that states, "Unable to load file associations".

If the specified file is in the correct driver replacement map file format, the debugger loads the file's contents and uses them as the driver replacement map. This map remains until you exit the debugger, or until you issue another `.kdfiles` command.

After the file has been read, the driver replacement map is not affected by subsequent changes to the file (unless these changes are followed by another `.kdfiles` command).

User Mode File Replacement

User Mode File Replacement was added in version 2004 of Windows. This support enables the following user mode files to be replaced with `.kdfiles`.

- User mode DLLs (also including NTDLL and KnownDLLs)
- User mode EXEs that are a main process image for CreateProcess

To use user mode `.kdfiles` support, you need to first enable kernel symbol loading using the `!gflag +ks1` debugger command or configure the ksl global flags in the registry. For more information about gflag, see [!gflag](#).

The following examples illustrate common usage.

```
dbgcmd
```

```
.kdfiles -m system32\userdll C:\myfiles\my_native_userdll.dll  
.kdfiles -m system32\userdll \\server\share\my_native_userdll.dll  
.kdfiles -m syswow64\ntdll.dll \\server\share\my_x86_wow64_ntdll.dll  
.kdfiles -m system32\userbase.dll \\server\share\my_native_userbase.dll
```

User mode .kdfiles ignores any failures to match a file and does not display an error message when a failure occurs.

Be careful to appropriately qualify the .kdfiles paths for user mode .kdfiles. It is a bad idea to just match ntdll.dll (instead of system32\ntdll.dll) as otherwise the Wow64 NTDLL will get replaced with the native one. Similar situations can arise with other ambiguous substring matches.

After build 20172, the user mode .kdfiles mechanism will attempt to pull files from the debugger until one attempt fails; then, the file name that failed to be pulled will not be tried again for the boot session, without manual intervention from the debugger to modify the target system state. On earlier builds, the user mode .kdfiles mechanism will make one attempt (whether successful or not) to pull a given file name per boot session. These policies reduce the overhead of communicating with the debugger for files that are not in the kdfiles list, or that are inaccessible for replacement, such as due to sharing violations from processes that may have already loaded a given file. Because of this behavior, it is generally advisable to configure any files to pull in the .kdfiles list up front, before they would first be referenced.

Be aware of limitations with being unable to replace already in use disk files, etc. As many system DLLs are won't be easily hot swappable after they have been loaded up initially, preset the gflags +ksl option and use .kdfiles to replace any user mode binaries right at boot.

For more information about enabling boot debugging, see [BCDEdit /bootdebug](#).

The use of the high speed/low latency KD transport KDNET is recommended to minimize system performance impacts.

Requirements

Version	Supported in Windows XP and later versions of the Windows operating system.
---------	---

.kdtargetmac (Display Target MAC Address)

Article • 10/25/2023

Display Target MAC Address

```
dbgcmd
```

```
.kdtargetmac
```

Parameters

Environment

Modes	kernel mode only
Targets	live debugging only
Platforms	all

Additional Information

Use the .kdtargetmac command to display the MAC (media access control) address of the target system.

```
dbgcmd
```

```
0: kd> .kdtargetmac
```

```
The target machine MAC address in open-device format is: XXXXXXXXXXXX
```

The .kdtargetmac command is available if KDNET is enabled on the target system. Use the BCDEdit command with the /dbgsettings option to display the configuration on the target system. A debugtype of *NET* indicates that KDNET is configured.

```
dbgcmd
```

```
C:\WINDOWS\system32>bcdedit /dbgsettings  
key 1.2.3.4  
debugtype NET  
hostip 192.168.1.10
```

```
port          50000
dhcp          Yes
The operation completed successfully.
```

For more information, see [Setting Up KDNET Network Kernel Debugging Manually](#).

Remarks

Knowing the MAC address of the target system can be useful for network tracing and other utilities.

.kframes (Set Stack Length)

Article • 10/25/2023

The **.kframes** command sets the default length of a stack trace display.

```
dbgcmd  
.kframes FrameCountDefault
```

Parameters

FrameCountDefault

Specifies the number of stack frames to display when a stack trace command is used.

Environment

Environment	
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

You can use the **.kframes** command to set the default length of a stack trace display. This length controls the number of frames that the **k**, **kb**, **kp**, **kP**, and **kv** commands display and the number of **DWORD_PTR**s that the **kd** command displays.

You can override this default length by using the *FrameCount* or *WordCount* parameters for these commands.

Use the **.kframes** command, without any parameters to see current value.

```
dbgcmd  
.kframes  
Default stack trace depth is 0x256 frames
```

Use the following command to set the stack trace depth to 0x2.

```
dbgcmd
```

```
.kframes 0x2  
Default stack trace depth is 0n2 frames
```

Use the following commands to display two stack and two raw stack values.

```
dbgcmd
```

```
k  
# Child-SP          RetAddr          Call Site  
00 00000054`b71ffb78 00007ffe`1ee672ae  ntdll!DbgBreakPoint  
01 00000054`b71ffb80 00007ffe`1e2a3e2d  ntdll!DbgUiRemoteBreakin+0x4e  
  
kd  
00000054`b71ffb70  00000000  
00000054`b71ffb74  00000000
```

Specify the *FrameCount* and *WordCount* to display additional values.

```
dbgcmd
```

```
k 3  
# Child-SP          RetAddr          Call Site  
00 00000054`b71ffb78 00007ffe`1ee672ae  ntdll!DbgBreakPoint  
01 00000054`b71ffb80 00007ffe`1e2a3e2d  ntdll!DbgUiRemoteBreakin+0x4e  
02 00000054`b71ffbb0 00007ffe`1eddef48  KERNEL32!BaseThreadInitThunk+0x1d  
  
kd 5  
00000054`b71ffb70  00000000  
00000054`b71ffb74  00000000  
00000054`b71ffb78  1ee672ae  
00000054`b71ffb7c  00007ffe  
00000054`b71ffb80  00000000
```

See also

[k, kb, kc, kd, kp, kP, kv \(Display Stack Backtrace\)](#)

.kill (Kill Process)

Article • 10/25/2023

In user mode, the `.kill` command ends a process that is being debugged.

In kernel mode, the `.kill` command ends a process on the target computer.

User-Mode Syntax

```
dbgcmd  
.kill [ /h | /n ]
```

Kernel-Mode Syntax

```
dbgcmd  
.kill Process
```

Parameters

/h

(User mode only) Any outstanding debug event will be continued and marked as handled. This is the default.

/n

(User mode only) Any outstanding debug event will be continued without being marked as handled.

Process

Specifies the address of the process to be terminated. If *Process* is omitted or zero, the default process for the current system state will be terminated.

Environment

In kernel mode, this command is supported on Microsoft Windows Server 2003 and later versions of Windows.

Modes	user mode, kernel mode
Targets	live debugging only

Remarks

In user mode, this command ends a process that is being debugged. If the debugger is attached to a child process, you can use `.kill` to end the child process without ending the parent process. For more information, see Examples.

In kernel mode, this command schedules the selected process on the target computer for termination. The next time that the target can run (for example, by using a [g \(Go\)](#) command), the specified process is ended.

You cannot use this command during local kernel debugging.

Examples

Using `.childdbg`

Suppose you attach a debugger to parent process (Parent.exe) before it creates a child process. You can enter the command [.childdbg 1](#) to tell the debugger to attach to any child process that the parent creates.

```
dbgcmd  
1:001> .childdbg 1  
Processes created by the current process will be debugged
```

Now let the parent process run, and break in after it has created the child process. Use the [| \(Process Status\)](#) command to see the process numbers for the parent and child processes.

```
dbgcmd  
0:002> |*  
. 0    id: 7f8 attach  name: C:\Parent\x64\Debug\Parent.exe  
     1    id: 2d4 child   name: notepad.exe
```

In the preceding output, the number of the child process (notepad.exe) is 1. The dot (.) at the beginning of the first line tells us that the parent process is the current process. To make the child process the current process, enter `|1s`.

```
dbgcmd
```

```
0:002> |1s
...
1:001> |*
# 0    id: 7f8 attach  name: C:\Parent\x64\Debug\Parent.exe
. 1    id: 2d4 child   name: notepad.exe
```

To kill the child process, enter the command `.kill`. The parent process continues to run.

```
dbgcmd

1:001> .kill
Terminated. Exit thread and process events will occur.
1:001> g
```

Using the `-o` parameter

When you start WinDbg or CDB, you can use the `-o` parameter to tell the debugger that it should attach to child processes. For example, the following command starts WinDbg, which starts and attaches to Parent.exe. When Parent.exe creates a child process, WinDbg attaches to the child process.

```
windbg -g -G -o Parent.exe
```

For more information, see [WinDbg Command-Line Options](#) and [CDB Command-Line Options](#).

Requirements

Version	Versions:(Kernel mode) Supported in Windows Server 2003 and later.
---------	--

.lastevent (Display Last Event)

Article • 10/25/2023

The `.lastevent` command displays the most recent exception or event that occurred.

```
dbgcmd
  .lastevent
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about exceptions and events, see [Controlling Exceptions and Events](#).

Remarks

Breaking into the debugger always creates an exception. There is always a *last event* when the debugger accepted command input. If you break into the debugger by using [CTRL+C](#), [CTRL+BREAK](#), or Debug | Break, an exception code of 0x80000003 is created.

.lines (Toggle Source Line Support)

Article • 10/25/2023

The `.lines` command enables or disables support for source-line information.

dbgcmd

```
.lines [-e|-d|-t]
```

Parameters

-e

Enables source line support.

-d

Disables source line support.

-t

Turns source line support on or off. If you do not specify parameters for `.lines`, the default behavior of the `.lines` command is this switching of source line support.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about source debugging and related commands, see [Debugging in Source Mode](#).

Remarks

You must enable source line support before you can perform source-level debugging. This support enables the debugger to load source line symbols.

You can enable source line support by using the `.lines` command or the [-lines command-line option](#). If source line support is already enabled, using the `.lines` command disables this support.

By default, if you do not use the `.lines` command, WinDbg turns on source line support, and console debuggers (KD, CDB, NTSD) turn off the support. For more information about how to change this setting, see [Setting Symbol Options](#).

.load, .loadby (Load Extension DLL)

Article • 10/25/2023

The **.load** and **.loadby** commands load a new extension DLL into the debugger.

```
dbgcmd  
.load DLLName  
!DLLName.load  
.loadby DLLName ModuleName
```

Parameters

DLLName

Specifies the debugger extension DLL to load. If you use the **.load** command, *DLLName* should include the full path. If you use the **.loadby** command, *DLLName* should include only the file name.

ModuleName

Specifies the module name of a module that is located in the same directory as the extension DLL that *DLLName* specifies.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to load, unload, and control extensions, see [Loading Debugger Extension DLLs](#).

Remarks

When you use the **.load** command, you must specify the full path.

When you use the **.loadby** command, you do not specify the path. Instead, the debugger finds the module that the *ModuleName* parameter specifies, determines the path of that module, and then uses that path when the debugger loads the extension DLL. If the debugger cannot find the module or if it cannot find the extension DLL, you receive an error message that specifies the problem. There does not have to be any relationship between the specified module and the extension DLL. Using the **.loadby** command is therefore simply a way to avoid typing a long path.

After the **.load** or **.loadby** command has been completed, you can access the commands that are stored in the loaded extension.

To load an extension DLL, you can do one of the following:

- Use the **.load** or **.loadby** command.
- Execute an extension by issuing the full **!DLLName.ExtensionCommand** syntax. If the debugger has not yet loaded *DLLName.dll*, it loads the DLL at this point if it is located in the current DLL search path.

Use the [.chain](#) command to display information about what has been loaded and the current DLL search path.

```
dbgcmd

0:000> .chain
Extension DLL search Path:
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\WINXP;C:\Program
Files (x86)\Windows Kits\10\Debuggers\x64\winext;C:\Program Files
(x86)\Windows Kits\10\Debuggers\x64\winext\arcade;C:\Program Files
(x86)\Windows Kits\10\Debuggers\x64\pri;C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64;
Extension DLL chain:
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\SOS.dll: image
4.8.4084.0, API 1.0.0, built Sun Nov 24 00:38:52 2019
```

For example, the managed code SOS.dll is not in search path for DLLs shown above, so use the **.load** command with a complete path to load that dll.

```
dbgcmd

0:000> .load C:\Windows\Microsoft.NET\Framework64\v4.0.30319\SOS.dll
```

.locale (Set Locale)

Article • 10/25/2023

The `.locale` command sets the locale or displays the current locale.

```
dbgcmd
```

```
  .locale [Locale]
```

Parameters

Locale

Specifies the locale that you want. If you omit this parameter, the debugger displays the current locale.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about locale, see the [setlocale routine reference page](#).

Remarks

The locale controls how Unicode strings are displayed.

The following examples show the `.locale` command.

```
dbgcmd
```

```
kd> .locale
Locale: C

kd> .locale E
Locale: English_United States.1252
```

```
kd> .locale c  
Locale: Catalan_Spain.1252
```

```
kd> .locale C  
Locale: C
```

.logappend (Append Log File)

Article • 10/25/2023

The **.logappend** command appends a copy of the events and commands from the [Debugger Command window](#) to the specified log file.

dbgcmd

```
.logappend [/u] [FileName]
```

Parameters

/u

Writes the log file in Unicode format. If you omit this parameter, the debugger writes the log file in ASCII (ANSI) format.

Note When you are appending to an existing log file, you should use the **/u** parameter only if you created the log file by using the **/u** option. Otherwise, your log file will contain ASCII and Unicode characters, which might make it more difficult to read.

FileName

Specifies the name of the log file. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify the path, the debugger uses the current directory. If you omit *FileName*, the debugger names the file Dbgeng.log.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you already have a log file open when you run the **.logappend** command, the debugger closes the log file. If you specify the name of a file that already exists, the

debugger appends new information to the file. If the file does not exist, the debugger creates it.

.logclose (Close Log File)

Article • 10/25/2023

The `.logclose` command closes any open log file.

```
dbgcmd
```

```
    .logclose
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.logfile (Display Log File Status)

Article • 10/25/2023

The `.logfile` command determines whether a log file exists and displays the file's status.

```
dbgcmd
```

```
    .logfile
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.logopen (Open Log File)

Article • 10/25/2023

The **.logopen** command sends a copy of the events and commands from the [Debugger Command window](#) to a new log file.

```
dbgcmd  
.logopen [Options] [FileName]  
.logopen /d
```

Parameters

Options Any of the following options:

/t

Appends the process ID with the current date and time to the log file name. This data is inserted after the file name and before the file name extension.

/u

Writes the log file in Unicode format. If you omit this option, the debugger writes the log file in ASCII (ANSI) format.

FileName

Specifies the name of the log file. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify a path, the debugger uses the current directory. If you omit *FileName*, the debugger names the file Dbgeng.log.

/d

Automatically chooses a file name based on the name of the target process or target computer and the state of the target. The file always has the .log file name extension.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

If you already have a log file open when you run the **.logopen** command, the debugger closes it. If you specify a file name that already exists, the file's contents are overwritten.

The **.logopen /t** command appends the process ID, date, and time to the log file name. In the following example, the process ID in hexadecimal is 0x02BC, the date is February 28, 2005, and the time is 9:05:50.935.

```
dbgcmd
```

```
0:000> .logopen /t c:\logs\myfile.txt
Opened log file 'c:\logs\myfile_02BC_2005-02-28_09-05-50-935.txt'
```

.netuse (Control Network Connections)

Article • 10/25/2023

The `.netuse` command adds a connection to a network share.

```
dbgcmd
```

```
.netuse /a "[Local]" "Remote" "[User]" "[Password]"
```

Parameters

`/a`

Adds a new connection. You must always use the `/a` switch.

Local

Specifies the drive letter to use for the connection. You must enclose *Local* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Remote

Specifies the UNC path of the share that is being connected. You must enclose *Remote* in quotation marks.

User

Specifies the user name of an account that is authorized to establish the connection. You must enclose *User* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Password

Specifies the password that is associated with the *User* account. You must enclose *Password* in quotation marks. If you omit this parameter, you must include an empty pair of quotation marks as the parameter.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The **.netuse** command behaves like the **net use** Microsoft MS-DOS command.

If you use **.netuse** during a remote debugging session, this command affects the debugging server, not the debugging client.

The following example shows this command.

```
dbgcmd
```

```
0:000> .netuse "m:" "\\myserver\myshare" "" ""
```

.noshell (Prohibit Shell Commands)

Article • 12/05/2024

The `.noshell` command prevents you from using [.shell](#) commands.

dbgcmd
<code>.noshell</code>

Environment

[+] Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the command shell and for other ways to disable shell commands, see [Using Shell Commands](#).

Remarks

If you use the `.noshell` command, you cannot use [.shell \(Command Shell\)](#) commands as long as the debugger is running, even if you start a new debugging session.

If you are performing remote debugging, this command is useful for security purposes.

ⓘ Important

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Feedback

Was this page helpful?

 Yes

 No

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

.noversion (Disable Version Checking)

Article • 10/25/2023

The **.noversion** command disables all version checking of extension DLLs.

```
dbgcmd
  .noversion
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The build number of extension DLLs should match the build number of the computer that you are debugging, because the DLLs are compiled and linked with dependencies on specific versions of data structures. If the versions do not match, you typically receive the following message.

```
Console
  *** Extension DLL(1367 Free) does not match target system(1552 Free)
```

.ocommand (Expect Commands from Target)

Article • 10/25/2023

The **.ocommand** command enables the target application to send commands to the debugger.

```
dbgcmd  
.ocommand String  
.ocommand -d  
.ocommand
```

Parameters

String

Specifies the command prefix string. *String* can include spaces, but you cannot use C-style control characters such as `\"` and `\n`. You can also enclose *String* in quotation marks. However, if *String* includes a semicolon, leading spaces, or trailing spaces, you must enclose *String* in quotation marks.

-d

Deletes the command prefix string.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For more information about [OutputDebugString](#) and other user-mode functions that communicate with a debugger, see the Microsoft Windows SDK documentation.

Remarks

If you use the `.ocommand` command without parameters, the debugger displays the current command prefix string. To clear the existing string, use `.ocommand -d`.

When you have set a command prefix string, any target output (such as the contents of an [OutputDebugString](#) command) is scanned. If this output begins with the command prefix string, the text of the output that follows the prefix string is treated as a debugger command string and is run. When this text is executed, the command string is not displayed.

The target can include an [echo \(Echo Comment\)](#) command in the output string if you want additional messages. Target output that does not begin with the prefix string is displayed in the typical manner.

After the commands within the command string have been executed, the target remains broken into the debugger, unless the final command is [g \(Go\)](#).

The comparison between the command prefix string and the target output is not case sensitive. (However, subsequent uses of `.ocommand` display the string that you entered with the case preserved).

For this example, assume that you enter the following command in the debugger.

```
dbgcmd  
0:000> .ocommand magiccommand
```

Then, the target application executes the following line.

```
dbgcmd  
OutputDebugString("MagicCommand kb;g");
```

The debugger recognizes the command string prefix and executes `kb;g` immediately.

However, the following line does not cause any commands to be executed.

```
dbgcmd  
OutputDebugString("Command on next line.\nmagiccommand kb;g");
```

There are no commands executed from the preceding example because the command string prefix is not at the beginning of the output, even though it does begin a new line.

Note You should choose a command string prefix that will not likely appear in any target output other than your own commands.

.nvload (NatVis Load)

Article • 10/25/2023

The .nvload command loads a NatVis file into the debugger environment. After the visualization is loaded, it will be used to render data defined in the visualization.

dbgcmd

```
.nvload FileName|ModuleName
```

Parameters

FileName | ModuleName

Specifies the NatVis file name or module name to load.

The **FileName** is the explicit name of a .natvis file to load. A fully qualified path can be used.

The **ModuleName** is the name of a module in the target process being debugged. All NatVis files which are embedded within the symbol file (PDB) of the named module name are loaded, if there are any available.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see [Create Custom Views of Native Objects](#).

See also

[dx \(Display NatVis Expression\)](#)

.nvlist (NatVis List)

Article • 10/25/2023

The .nvllist command lists the NatVis files loaded into the debugger environment.

```
dbgcmd
```

```
.nvlist
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see [Create Custom Views of Native Objects](#).

See also

[dx \(Display NatVis Expression\)](#)

.nvunload (NatVis Unload)

Article • 10/25/2023

The .nvunload command unloads a NatVis file from the debugger environment.

dbgcmd

```
.nvunload FileName|ModuleName
```

FileName | ModuleName

Specifies the NatVis file name or module name to unload.

The **FileName** is the explicit name of a .natvis file to unload. A fully qualified path can be used.

The **ModuleName** is the name of a module in the target process being debugged. All NatVis files which are embedded within the symbol file (PDB) of the named module name are unloaded.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see [Create Custom Views of Native Objects](#).

See also

[dx \(Display NatVis Expression\)](#)

.nvunloadall (NatVis Unload All)

Article • 10/25/2023

The .nvunloadall command unloads all of the NatVis files from the debugger environment.

```
dbgcmd  
.nvunloadall
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information, see [Create Custom Views of Native Objects](#).

See also

[dx \(Display NatVis Expression\)](#)

.ofilter (Filter Target Output)

Article • 10/25/2023

The `.ofilter` command filters the output from the target application or target computer.

```
dbgcmd  
.ofilter [!/] String  
.ofilter ""  
.ofilter
```

Parameters

`!`

Reverses the filter so that the debugger displays only output that does not contain *String*. If you do not use this parameter, the debugger displays only output that contains *String*.

String

Specifies the string to match in the target's output. *String* can include spaces, but you cannot use C-style control characters such as `\"` and `\n`. *String* might contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#).

You can enclose *String* in quotation marks. However, if *String* includes a semicolon, leading spaces, or trailing spaces, you must use quotation marks. Alphanumeric characters in *String* are converted to uppercase letters, but the actual pattern matching is case insensitive.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about [OutputDebugString](#) and other user-mode routines, see the Microsoft Windows SDK documentation. For more information about [DbgPrint](#), [DbgPrintEx](#), and other kernel-mode routines, see the Windows Driver Kit (WDK).

Remarks

If you use the `.ofilter` command without parameters, the debugger displays the current pattern-matching criteria.

To clear the existing filter, use `.ofilter ""`. This command filters any data that is sent by user-mode routines (such as [OutputDebugString](#)) and kernel-mode routines (such as [DbgPrint](#)). However, the debugger always displays prompts that [DbgPrompt](#) sends.

The [DbgPrintEx](#) and [KdPrintEx](#) routines supply another method of filtering debugging messages that you do not want.

.open (Open Source File)

Article • 10/25/2023

The **.open** command searches the source path for a source file and opens this file.

```
dbgcmd  
.open [-m Address] FileName  
.open -a Address
```

Parameters

FileName

Specifies the source file name. This name can include an absolute or relative path. Unless you specify an absolute path, the path is interpreted as relative to a directory in the source path.

-m ** Address**

Specifies an address within the source file. This address must be contained in a known module. You should use the **-m **** Address** parameter if the file that *FileName* specifies is not unique. For more information about the syntax, see [Address and Address Range Syntax](#).

The **-m** parameter is required if you are using a [source server](#) to retrieve the source files.

-a Address

Specifies an address within the source file. This address must be contained in a known module. If the debugger can find the source file, the debugger loads and opens the file, and the line that corresponds to the specified address is highlighted. If the debugger cannot find the source file, the address is displayed in the [Disassembly window](#). For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

You can use the **.open** command only in WinDbg, and you cannot use it in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump

Item	Description
Platforms	All

Additional Information

For more information about source files and source paths and for other ways to load source files, see [Source Path](#).

.opendump (Open Dump File)

Article • 10/25/2023

The **.opendump** command opens a dump file for debugging.

dbgcmd

```
.opendump DumpFile  
.opendump /c "DumpFileInArchive" [CabFile]
```

Parameters

DumpFile

Specifies the name of the dump file to open. *DumpFile* should include the file name extension (typically .dmp or .mdmp) and can include an absolute or relative path. Relative paths are relative to the directory that you started the debugger in.

/c "DumpFileInArchive"

Specifies the name of a dump file to debug. This dump file must be contained in the archive file that *CabFile* specifies. You must enclose the *DumpFileInArchive* file in quotation marks.

CabFile

Specifies the name of an archive file to open. *CabFile* should include the file name extension (typically .cab) and can include an absolute or relative path. Relative paths are relative to the directory that you started the debugger in. If you use the */c* switch to specify a dump file in an archive but you omit *CabFile*, the debugger reuses the archive file that you most recently opened.

Environment

Modes	User mode, kernel mode
Targets	Crash dump only (but you can use this command if other sessions are running)
Platforms	All

Remarks

After you use the `.opendump` command, you must use the **g (Go)** command to finish loading the dump file.

When you are opening an archive file (such as a CAB file), you should use the `/c` switch. If you do not use this switch and you specify an archive for *DumpFile*, the debugger opens the first file that has an `.mdmp` or `.dmp` file name extension within this archive.

You can use `.opendump` even if a debugging session is already in progress. This feature enables you to debug more than one crash dump at the same time. For more information about how to control a multiple-target session, see [Debugging Multiple Targets](#).

Note There are complications, when you debug live targets and dump targets together, because commands behave differently for each type of debugging. For example, if you use the **g (Go)** command when the current system is a dump file, the debugger begins executing, but you cannot break back into the debugger, because the break command is not recognized as valid for dump file debugging.

.outmask (Control Output Mask)

Article • 10/25/2023

The **.outmask** command controls the current output mask.

```
dbgcmd  
.outmask[-] [/I] Expression  
.outmask /a  
.outmask /d
```

Parameters

Expression

Specifies the flags to add to the mask. *Expression* can be any ULONG value that specifies the flag bits that you want. For a list of the possible flags, see the table in the Remarks section.

-

Removes the bits that *Expression* specifies from the mask, instead of adding them to the mask.

/I

Preserves the current value of the log file's output mask. If you do not include /I, the log file's output mask is the same as the regular output mask.

/a

Activates all mask flags. This parameter is equivalent to **.outmask 0xFFFFFFFF**.

/d

Restores the output mask to the default value. This parameter is equivalent to **.outmask 0x3F7**.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Each output mask flag enables the debugger to display certain output in the [Debugger Command Window](#). If all of the mask flags are set, all output is displayed.

You should remove output mask flags with caution, because you might be unable to read debugger output.

The following flag values exist.

Value	Default setting	Description
1	On	Normal output
2	On	Error output
4	On	Warnings
8	Off	Additional output
0x10	On	Prompt output
0x20	On	Register dump before prompt
0x40	On	Warnings that are specific to extension operation
0x80	On	Debug output from the target (for example, <code>OutputDebugString</code> or <code>DbgPrint</code>)
0x100	On	Debug input expected by the target (for example, <code>DbgPrompt</code>)
0x200	On	Symbol messages (for example, <code>!sym noisy</code>)

.pagein (Page In Memory)

Article • 10/25/2023

The `.pagein` command pages in the specified region of memory.

dbgcmd

```
.pagein [Options] Address
```

Parameters

Options

Any of the following options:

/p *Process*

Specifies the address of the process that owns the memory that you want to page in. (More precisely, this parameter specifies the address of the EPROCESS block for the process.) If you omit *Process* or specify zero, the debugger uses the current process setting. For more information about the process setting, see [.process \(Set Process Context\)](#)

/f

Forces the memory to be paged in, even if the address is in kernel memory and the version of Windows does not support this action.

Address

Specifies the address to page in.

Environment

Item	Description
Modes	Kernel mode only (but not during local kernel debugging)
Targets	Live debugging only
Platforms	All

Remarks

After you run the `.pagein` command, you must use the [g \(Go\)](#) command to resume program execution. After a brief time, the target computer automatically breaks into the debugger again.

At this point, the address that you specify is paged in. If you use the `/p` option, the process context is also set to the specified process, exactly as if you used the [.process /i Process](#) command.

If the address is already paged in, the `.pagein` command still checks that the address is paged in and then breaks back into the debugger. If the address is invalid, this command only breaks back into the debugger.

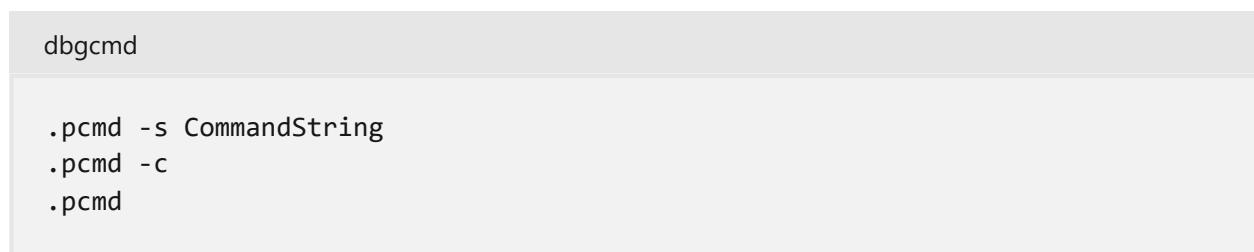
Requirements

Supported in Windows XP and later versions of Windows.

.pcmd (Set Prompt Command)

Article • 10/25/2023

The **.pcmd** command causes the debugger to issue a command whenever the target stops executing and to display a prompt in the [Debugger Command window](#) with register or target state information.



Parameters

-s ** *CommandString***

Specifies a new prompt command string. Whenever the target stops executing, the debugger issues and immediately runs the *CommandString* command. If *CommandString* contains spaces or semicolons, you must enclose it in quotation marks.

-c

Deletes any existing prompt command string.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the Debugger Command window prompt, see [Using Debugger Commands](#).

Remarks

If you use the `.pcmd` command without parameters, the current prompt command is displayed.

When you set a prompt command by using `.pcmd -s`, the specified *CommandString* is issued whenever the target stops executing (for example, when a `g`, `p`, or `t` command ends). The *CommandString* command is not issued when you use a non-execution command, unless that command displays registers or target state information.

In the following example, the first use of `.pcmd` sets a fixed string that appears with the prompt. The second use of `.pcmd` causes the debugger to display the target's current process ID and thread ID every time that the prompt appears. The special prompt does not appear after the `.ttime` command is used, because that command does not involve execution.

```
dbgcmd

0:000> .pcmd
No per-prompt command

0:000> .pcmd -s ".echo Execution is done."
Per-prompt command is '.echo Execution is done.'

0:000> t
Prymes!isPrime+0xd0:
004016c0 837dc400      cmp dword ptr [ebp-0x3c],0x0
ss:0023:0012fe70=00000002
Execution is done.

0:000> t
Prymes!isPrime+0xd4:
004016c4 7507          jnz    Prymes!isPrime+0xdd (004016cd)
[br=1]
Execution is done.

0:000> .ttime
Created: Thu Aug 21 13:18:59 2003
Kernel: 0 days 0:00:00.031
User:   0 days 0:00:00.000

0:000> .pcmd -s "r $tpid, $tid"
Per-prompt command is 'r $tpid, $tid'

0:000> t
Prymes!isPrime+0xdd:
004016cd ebc0          jmp    Prymes!isPrime+0x9f (0040168f)
$tpid=0000080c $tid=00000514

0:000> t
Prymes!isPrime+0x9f:
0040168f 8b55fc        mov    edx,[ebp-0x4]
```

ss:0023:0012fea8=00000005
\$tpid=0000080c \$tid=00000514

.pop (Restore Debugger State)

Article • 10/25/2023

The **.pop** command restores the state of the debugger to a state that has previously been saved by using the [.push \(Save Debugger State\)](#) command.

```
dbgcmd  
.pop  
.pop /r  
.pop /r /q
```

Parameters

/r

Specifies that the saved values of the pseudo-registers \$t0 to \$t19 should be restored. If **/r** is not included, these values are not affected by the **.pop** command.

/q

Specifies that the command executes quietly. That is, the command executes without displaying any output.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command is most useful when used with [scripts](#) and [debugger command programs](#) so that they can work with one fixed state. If the command is successful, no output is displayed.

.prefer_dml (Prefer Debugger Markup Language)

Article • 10/25/2023

The `.prefer_dml` command sets the default behavior for commands that are capable of providing output in the Debugger Markup Language (DML) format.

```
dbgcmd  
.prefer_dml 0  
.prefer_dml 1
```

Parameters

0

By default, all commands will provide plain text output.

1

By default, commands that are capable of providing DML output will provide DML output.

See also

[Debugger Markup Language Commands](#)

.process (Set Process Context)

Article • 10/25/2023

The **.process** command specifies which process is used for the process context.

dbgcmd

```
.process [/i] [/p [/r]] [/P] [Process]
```

Parameters

/i

Live debugging only; not during local kernel debugging) Specifies that *Process* is to be debugged *invasively*. This kind of debugging means that the operating system of the target computer actually makes the specified process active. (Without this option, the **.process** command alters the debugger's output but does not affect the target computer itself.) If you use **/i**, you must use the [g \(Go\)](#) command to execute the target. After several seconds, the target breaks back in to the debugger, and the specified *Process* is active and used for the process context.

/p

Translates all transition page table entries (PTEs) for this process to physical addresses before access, if you use **/p** and *Process* is nonzero. This translation might cause slowdowns, because the debugger must find the physical addresses for all of the memory that this process uses. Also, the debugger might have to transfer a significant amount of data across the debug cable. (This behavior is the same as [.cache forcedecodeuser](#).)

If you include the **/p** option and *Process* is zero or you omit it, the translation is disabled. (This behavior is the same as [.cache noforcedecodeptes](#).)

/r

Reloads user-mode symbols after the process context has been set, if you use the **/r** and **/p** options. (This behavior is the same as [.reload /user](#).)

/P

(Live debugging and complete memory dumps only) Translates all transition page table entries (PTEs) to physical addresses before access, if you use **/P** and *Process* is nonzero. Unlike the **/p** option, the **/P** option translates the PTEs for all user-mode and kernel-mode processes, not only the specified process. This translation might cause slowdowns, because the debugger must find the physical addresses for all memory in use. Also, the

debugger might have to transfer lots of data across the debug cable. (This behavior is the same as [.cache forcedecodeptes](#).)

Process

Specifies the address of the process that you want. (More precisely, this parameter specifies the address of the EPROCESS block for this process). The process context is set to this process. If you omit *Process* or specify zero, the process context is reset to the default process for the current system state. (If you used the /i option to set process context, you must use the /i option to reset the process context.)

Environment

Modes	Kernel mode only
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about the process context and other context settings, see [Changing Contexts](#).

Remarks

Typically, when you are doing kernel debugging, the only visible user-mode address space is the one that is associated with the current process.

The **.process** command instructs the kernel debugger to use a specific user-mode process as the *process context*. This usage has several effects, but the most important is that the debugger has access to the virtual address space of this process. The debugger uses the page tables for this process to interpret all user-mode memory addresses, so you can read and write to this memory.

The **.context (Set User-Mode Address Context)** command has a similar effect. However, the **.context** command sets the *user-mode address context* to a specific page directory, while the **.process** command sets the process context to a specific process. On an x86-based processor, **.context** and **.process** have almost the same effect. However, on an Itanium-based processor, a single process might have more than one page directory. In this situation, the **.process** command is more powerful, because it enables access to all of the page directories that are associated with a process. For more information about the process context, see [Process Context](#).

Note If you are performing live debugging, you should use the **/i** or the **/p** parameter. Without one of these parameters, you cannot correctly display user-mode or session memory.

The **/i** parameter activates the target process. When you use this option, you must execute the target once for this command to take effect. If you execute again, the process context is lost.

The **/p** parameter enables the **forcedecodeuser** setting. (You do not have to use **/p** if the **forcedecodeuser** option is already active.) The process context and the **forcedecodeuser** state remain only until the target executes again.

If you are performing crash dump debugging, the **/i** and **/p** options are not available. However, you cannot access any part of the user-mode process' virtual address space that were paged to disk when the crash occurred.

If you want to use the kernel debugger to set breakpoints in user space, use the **/i** option to switch the target to the correct process context.

The following example shows how to use the **!process** extension to find the address of the EPROCESS block for the desired process.

```
dbgcmd

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0 SessionId: 0 Cid: 0008 Peb: 00000000 ParentCid: 0000
    DirBase: 00030000 ObjectTable: fe529b68 TableSize: 50.
    Image: System

.....
PROCESS fe3c0d60 SessionId: 0 Cid: 0208 Peb: 7ffdf000 ParentCid: 00d4
    DirBase: 0011f000 ObjectTable: fe3d0f48 TableSize: 30.
    Image: regsvc.exe
```

Now the example uses the **.process** command with this process address.

```
dbgcmd

kd> .process fe3c0d60
Implicit process is now fe3c0d60
```

Notice that this command makes the **.context** command unnecessary. The user-mode address context already has the desired value.

```
dbgcmd
```

```
kd> .context  
User-mode page directory base is 11f000
```

This value enables you to examine the address space in various ways. For example, the following example shows the output of the [!peb](#) extension.

```
dbgcmd
```

```
kd> !peb  
PEB at 7FFDF000  
InheritedAddressSpace: No  
ReadImageFileExecOptions: No  
BeingDebugged: No  
ImageBaseAddress: 01000000  
Ldr.Initialized: Yes  
Ldr.InInitializationOrderModuleList: 71f40 . 77f68  
Ldr.InLoadOrderModuleList: 71ec0 . 77f58  
Ldr.InMemoryOrderModuleList: 71ec8 . 77f60  
    01000000 C:\WINNT\system32\regsvc.exe  
    77F80000 C:\WINNT\System32\ntdll.dll  
    77DB0000 C:\WINNT\system32\ADVAPI32.dll  
    77E80000 C:\WINNT\system32\KERNEL32.DLL  
    77D40000 C:\WINNT\system32\RPCRT4.DLL  
    77BE0000 C:\WINNT\system32\secur32.dll  
SubSystemData: 0  
ProcessHeap: 70000  
ProcessParameters: 20000  
    WindowTitle: 'C:\WINNT\system32\regsvc.exe'  
    ImageFile: 'C:\WINNT\system32\regsvc.exe'  
    CommandLine: 'C:\WINNT\system32\regsvc.exe'  
    DllPath:  
'C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;C:\WINNT;C:\WINNT\sys  
tem32;C:\WINNT;C:\WINNT\System32\Wbem;C:\PROGRA~1\COMMON~1\AUTODE~1'  
    Environment: 0x10000
```

.prompt_allow (Control Prompt Display)

Article • 10/25/2023

The `.prompt_allow` command controls what information is displayed during stepping and tracing and whenever the target's execution stops.

```
dbgcmd  
.prompt_allow {+|-}Item [...]  
.prompt_allow
```

Parameters

+

Displays the specified item at the stepping, tracing, and execution prompt. You must add a space before the plus sign (+), but you cannot add a space after it.

-

Prevents the specified item from being displayed at the stepping, tracing, and execution prompt. You must add a space before the minus sign (-), but you cannot add a space after it.

Item

Specifies an item to display or not display. You can specify any number of items.

Separate multiple items by spaces. You must add a plus sign (+) or minus sign (-) before each item. You can use the following items:

dis

The disassembled instructions at the current location.

ea

The effective address for the current instruction.

reg

The current state of the most important registers. You can disable register display by using the [pr](#), [tr](#), or `.prompt_allow -reg` command. All three of these commands control the same setting, and you can use any of them to override any previous use of these commands.

You can also disable register display by using the `l-os` command. This setting is separate from the other three commands, as described in the following Remarks section. To control which registers and flags are displayed, use the [rm \(Register Mask\)](#) command.

src

The source line that corresponds to the current instruction. You can disable source line display by using the l+ls or .prompt_allow -src; commands. You must enable source line display through both mechanisms to be visible.

sym

The symbol for the current instruction. This symbol includes the current module, function name, and offset.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about commands that affect execution, see [Controlling the Target](#).

Remarks

You can use the **.prompt_allow** command without parameters to show which items are displayed and are not displayed. Every time that you run **.prompt_allow**, the debugger changes the status of only the specified items.

By default, all items are displayed.

If you have used the l+os option, this option overrides any of the **.prompt_allow** options other than **src**.

You can also use a complex command such as the following example.

```
dbgcmd

0:000> .prompt_allow -reg -dis +ea
Allow the following information to be displayed at the prompt:
(Other settings can affect whether the information is actually displayed)
    sym - Symbol for current instruction
    ea - Effective address for current instruction
    src - Source info for current instruction
Do not allow the following information to be displayed at the prompt:
```

dis - Disassembly of current instruction
reg - Register state

.push (Save Debugger State)

Article • 10/25/2023

The **.push** command saves the current state of the debugger.

```
dbgcmd  
.push  
.push /r  
.push /r /q
```

Parameters

/r

Specifies that the current values in the pseudo-registers **\$t0** to **\$t19** should be saved. If the **/r** parameter is not used, these values are not saved by the **.push** command.

/q

Specifies that the command executes quietly. That is, the command executes without displaying any output.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command is most useful when used with [scripts](#) and [debugger command programs](#) so that they can work with one fixed state. To restore the debugger to a state that was previously saved using this command, use the [.pop \(Restore Debugger State\)](#) command. If the command is successful, no output is displayed.

.quit_lock (Prevent Accidental Quit)

Article • 12/05/2024

The **.quit_lock** command sets a password to prevent you from accidentally ending the debugging session.

dbgcmd

```
.quit_lock /s NewPassword  
.quit_lock /q Password  
.quit_lock
```

Parameters

/s ** *NewPassword***

Prevents the debugging session from ending and stores *NewPassword*. You cannot end the debugger session until you use the **.quit_lock /q** command together with this same password. *NewPassword* can be any string. If it contains spaces, you must enclose *NewPassword* in quotation marks.

/q ** *Password***

Enables the debugging session to end. *Password* must match the password that you set with the **.quit_lock /s** command.

Environment

[+] [Expand table](#)

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Without parameters, **.quit_lock** displays the current lock status, including the full text of the password.

You can repeat the `.quit_lock /s` command to change an existing password.

When you use `.quit_lock /q`, the lock is removed. This command does not close the debugger. Instead, the command only enables you to exit the session in the typical manner when you want to.

Note The password is not "secret". Any remote user who is attached to the debugging session can use `.quit_lock` to determine the password. The purpose of this command is to prevent accidental use of the [q \(Quit\)](#) command. This command is especially useful if restarting the debugging session might be difficult (for example, during remote debugging).

You cannot use the `.quit_lock /s` command in [Secure Mode](#). If you use this command before Secure Mode is activated, the password protection remains, but you cannot change or remove the password.

Important

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Feedback

Was this page helpful?

 Yes

 No

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

.readmem (Read Memory from File)

Article • 10/25/2023

The **.readmem** command reads raw binary data from the specified file and copies the data to the target computer's memory.

dbgcmd

```
.readmem FileName Range
```

Parameters

FileName

Specifies the name of the file to read. You can specify a full path or only the file name. If the file name contains spaces, enclose *FileName* in quotation marks. If you do not specify a path, the debugger uses the current directory.

Range

Specifies the address range for putting the data in memory. This parameter can contain a starting and ending address or a starting address and an object count. For more information about the syntax, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The memory data is copied literally to the target computer. The debugger does not parse the data in any way. For example, the **.readmem myfile 1000 10** command copies 10 bytes from the Myfile file and stores them in the target computer's memory, starting at address 1000.

The **.readmem** command is the opposite of the [.writemem \(Write Memory to File\)](#) command.

.reboot (Reboot Target Computer)

Article • 10/25/2023

The `.reboot` command restarts the target computer.

```
dbgcmd
```

```
.reboot
```

Environment

Modes	Kernel mode only
Targets	Live debugging only
Platforms	All

Additional Information

For more information about related commands and an explanation of how the restart process affects the debugger, see [Crashing and Rebooting the Target Computer](#).

.record_branches (Enable Branch Recording)

Article • 10/25/2023

The **.record_branches** command enables the recording of branches that the target's code executed.

dbgcmd

```
.record_branches {1|0}  
.record_branches
```

Environment

Modes	User mode, kernel mode
Targets	Live debugging only
Platforms	x64-based only

Remarks

The **.record_branches 1** command enables the recording of branches in the target's code. The **.record_branches 0** command disables this recording.

Without parameters, **.record_branches** displays the current status of this setting.

.reload (Reload Module)

Article • 10/25/2023

The **.reload** command deletes all symbol information for the specified module and reloads these symbols as needed. In some cases, this command also reloads or unloads the module itself.

dbgcmd

```
.reload [Options] [Module[=Address[,Size[,Timestamp]]]]  
.reload -?
```

Parameters

Options Any of the following options:

/d

Reloads all modules in the debugger's module list. (When you omit all parameters, this situation is the default during user-mode debugging.)

/f

Forces the debugger to immediately load the symbols. This parameter overrides *lazy symbol loading*. For more information, see the following Remarks section.

/i

Ignores a mismatch in the .pdb file versions. (If you do not include this parameter, the debugger does not load mismatched symbol files.) When you use **/i**, **/f** is used also, even if you do not explicitly specify it.

/l

Lists the modules but does not reload their symbols. (In kernel mode, this parameter provides output similar to the **!m** command.)

/n

Reloads kernel symbols only. This parameter does not reload any user symbols. (You can use this option only during kernel-mode debugging.)

/o

Forces the cached files in a symbol server's downstream store to be overwritten. When you use this flag, you should also include **/f**. By default, the downstream store files are never overwritten.

Because the symbol server uses distinct file names for the symbols of every different build of a binary, you do not have to use this option unless you believe your downstream store has become corrupted.

/s

Reloads all modules in the system's module image list. (When you omit all parameters, this situation is the default during kernel-mode debugging.)

If you are loading an individual system module by name while you perform user-mode debugging, you must include /s.

/u

Unloads the specified module and all its symbols. The debugger unloads any loaded module whose name matches *Module*, regardless of the full path. Image names are also searched. For more information, see the note in the following Remarks section.

/unl

Reloads symbols based on the image information in the unloaded module list.

/user

Reloads user symbols only. (You can use this option only during kernel-mode debugging.)

/v

Turns on verbose mode.

/w

Treats *Module* as a literal string. This treatment prevents the debugger from expanding wildcard characters.

Module

Specifies the name of an image on the target system for which to reload symbols on the host computer. *Module* should include the name and file name extension of the file. Unless you use the /w option, *Module* might contain a variety of wildcard characters and specifiers. For more information about the syntax, see [String Wildcard Syntax](#). If you omit *Module*, the behavior of the .reload command depends on which *Options* you use.

Address

Specifies the base address of the module. Typically, you have to have this address only if the image header has been corrupted or is paged out.

Size

Specifies the size of the module image. In many situations, the debugger knows the correct size of the module. When the debugger does not know the correct size, you

should specify *Size*. This size can be the actual module size or a larger number, but the size should not be a smaller number. Typically, you have to have this size only if the image header has been corrupted or is paged out.

Timestamp

Specifies the timestamp of the module image. In many situations, the debugger knows the correct timestamp of the module. When the debugger does not know the timestamps, you should specify *Timestamp*. Typically, you have to have this timestamp only if the image header has been corrupted or is paged out.

Note There must be no blank space between the *Address*, *Size*, and *Timestamp* parameters.

-?

Displays a short help text for this command.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about deferred (lazy) symbol loading, see [Deferred Symbol Loading](#). For more information about other symbol options, see [Setting Symbol Options](#).

Remarks

The `.reload` command does not cause symbol information to be read. Instead, this command lets the debugger know that the symbol files might have changed or that a new module should be added to the module list. This command causes the debugger to revise its module list and delete its symbol information for the specified modules. The actual symbol information is not read from the individual .pdb files until the information is needed. (This kind of loading is known as *lazy symbol loading* or *deferred symbol loading*.)

You can force symbol loading to occur by using the `/f` option or by issuing an [!ld \(Load Symbols\)](#) command.

The **.reload** command is useful if the system stops responding (that is, crashes), which might cause you to lose symbols for the target computer that is being debugged. The command can also be useful if you have updated the symbol tree.

If the image header is incorrect for some reason, such as the module being unloaded, or is paged out, you can load symbols correctly by using the **/unl** argument, or specifying both *Address* and *Size*.

The **.reload /u** command performs a broad search. The debugger first tries to match *Module* with an exact module name, regardless of path. If the debugger cannot find this match, *Module* is treated as the name of the loaded image. For example, if the HAL that resides in memory has the module name of halacpi.dll, both of the following commands unload its symbols.

```
dbgcmd  
kd> .reload /u halacpi.dll  
kd> .reload /u hal
```

If you are performing user-mode debugging and want to load a module that is not part of the target application's module list, you must include the **/s** option, as the following example shows.

```
dbgcmd  
0:000> .reload /u ntdll.dll  
Unloaded ntdll.dll  
0:000> .reload /s /f ntdll.dll
```

.remote (Create Remote.exe Server)

Article • 12/05/2024

The **.remote** command starts a [Remote.exe Server](#), enabling a remote connection to the current debugging session.

```
dbgcmd  
.remote session
```

Parameters

session

Specifies a name that you give to the debugging session.

Environment

You can use the **.remote** command in KD and CDB, but you cannot use it in WinDbg.

[+] Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about how to use Remote.exe Servers and Remote.exe Clients, see [Remote Debugging Through Remote.exe](#).

ⓘ Important

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Remarks

The `.remote` command creates a `Remote.exe` process and turns the current debugger into a `Remote.exe` Server. This server enables a `Remote.exe` Client to connect to the current debugging session.

Feedback

Was this page helpful?

 Yes

 No

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

.remote_exit (Exit Debugging Client)

Article • 12/05/2024

The **.remote_exit** command exits the debugging client but does not end the debugging session.

```
dbgcmd
```

```
.remote_exit [FinalCommands]
```

Parameters

FinalCommands

Specifies a command string to pass to the debugging server. You should separate multiple commands by using semicolons. These commands are passed to the debugging server and the connection is then broken.

Environment

You can use the **.remote_exit** command only in a script file. You can use it in KD and CDB, but you cannot use it in WinDbg.

[+] Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information about script files, see [Using Script Files](#). For more information about debugging clients and debugging servers, see [Remote Debugging Through the Debugger](#).

 **Important**

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Remarks

If you are using KD or CDB directly, instead of using a script, you can exit from the debugging client by using the **CTRL+B** key.

You cannot exit from a debugging client through a script that is executed in WinDbg.

Feedback

Was this page helpful?

 Yes

 No

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

.restart (Restart Target Application)

Article • 10/25/2023

The `.restart` command restarts the target application.

Do not confuse this command with the [.restart \(Restart Kernel Connection\)](#) command, which works only in kernel mode.

```
dbgcmd
```

```
.restart
```

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For more information about how to issue this command and an overview of related commands, see [Controlling the Target](#).

Remarks

CDB and WinDbg can restart a target application if the debugger originally created the application. You can use the `.restart` command even if the target application has already closed.

However, if the application is running and the debugger is later attached to the process, the `.restart` command has no effect.

After the process is restarted, it immediately breaks into the debugger.

In WinDbg, use the [View | WinDbg Command Line](#) command if you started your target from the WinDbg command prompt and you want to review this command line before you decide whether to use `.restart`.

.restart (Restart Kernel Connection)

Article • 04/03/2024

The **.restart** command restarts the kernel connection.

Do not confuse this command with the [.restart \(Restart Target Application\)](#) command, which works only in user mode.

```
dbgcmd
```

```
.restart
```

Environment

You can use the **.restart** command only in KD.

[] [Expand table](#)

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about reestablishing contact with the target, see [Synchronizing with the Target Computer](#).

Remarks

The **.restart** command is similar to the [CTRL+R \(Re-synchronize\)](#) command, except that **.restart** is even more extensive in its effect. This command is equivalent to ending the debugger and then attaching a new debugger to the target computer.

The **.restart** command is most useful when you are performing [remote debugging through remote.exe](#) and ending and restarting the debugger might be difficult. However, you cannot use **.restart** from a debugging client if you are performing remote debugging through the debugger.

Feedback

Was this page helpful?

 Yes

 No

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

.rrestart (Register for Restart)

Article • 10/25/2023

The **.rrestart** command registers the debugging session for restart in case of a reboot or an application failure.

```
dbgcmd
```

```
.rrestart
```

Remarks

This command does not work for elevated debugger sessions.

See also

[.urestart](#)

.scroll_prefs (Control Source Scrolling Preferences)

Article • 10/25/2023

The `.scroll_prefs` command controls the positioning of the source in a Source window when scrolling to a line.

```
dbgcmd  
.scroll_prefs Before After  
.scroll_prefs
```

Parameters

Before

Specifies how many source lines before the line you are scrolling to should be visible in the Source window.

After

Specifies how many source lines after the line you are scrolling to should be visible in the Source window.

Environment

This command is available only in WinDbg and cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

When this command is used with no parameters, the current source scrolling preferences are displayed.

.scriptdebug (Debug JavaScript)

Article • 10/25/2023

Use the `.scriptdebug` command to debug JavaScript scripts.

```
dbgcmd  
.scriptdebug FileName
```

Parameters

FileName

Specifies the name of the debugger JavaScript script to debug.

Environment

Modes	all
Targets	all
Platforms	all

Additional Information

For an overview of JavaScript debugging, see [JavaScript Debugger Scripting - JavaScript Debugging](#).

 **Note**

To use JavaScript Debugging with WinDbg, run the debugger as Administrator.

Remarks

Before you debug a JavaScript completed the following steps.

1. Load the sample script.

```
dbgcmd
```

```
0:000> .scriptload C:\MyScripts\DebuggableSample.js
```

To start actively debugging the script use the `.scriptdebug` command.

```
dbgcmd
```

```
0:000> .scriptdebug C:\MyScripts\DebuggableSample.js
>>> ***** DEBUGGER ENTRY DebuggableSample *****
      No active debug event!

>>> Debug [DebuggableSample <No Position>] >
```

Once you see the prompt `>>> Debug [DebuggableSample <No Position>] >` and a request for input, you are inside the script debugger.

Use the `.help` command or `?` to display a list of commands in the JavaScript debugging environment.

```
dbgcmd
```

```
>>> Debug [DebuggableSample <No Position>] >.help
Script Debugger Commands (*NOTE* IDs are **PER SCRIPT**):
  ? ..... Get help
  ? <expr> ..... Evaluate expression <expr> and
display result
  ?? <expr> ..... Evaluate expression <expr> and
display result
  | ..... List available scripts
  |<scriptid>s ..... Switch context to the given script
  bc \<bpid\> ..... Clear breakpoint by specified \
<bpid\>
  bd \<bpid\> ..... Disable breakpoint by specified \
<bpid\>
  be \<bpid\> ..... Enable breakpoint by specified \
<bpid\>
  bl ..... List breakpoints
  bp <line>:<column> ..... Set breakpoint at the specified
line and column
  bp <function-name> ..... Set breakpoint at the (global)
function specified by the given name
  bpc ..... Set breakpoint at current location
  dv ..... Display local variables of current
frame
  g ..... Continue script
  gu ..... Step out
  k ..... Get stack trace
  p ..... Step over
  q ..... Exit script debugger (resume
execution)
  sx ..... Display available events/exceptions
```

```
to break on
  sxe <event> ..... Enable break on <event>
  sxd <event> ..... Disable break on <event>
  t ..... Step in
  .attach <scriptId> ..... Attach debugger to the script
specified by <scriptId>
  .detach [<scriptId>] ..... Detach debugger from the script
specified by <scriptId>
  .frame <index> ..... Switch to frame number <index>
  .f+ ..... Switch to next stack frame
  .f- ..... Switch to previous stack frame
  .help ..... Get help
```

Events

Use the **sx** script debugger command to see the list of events that can be trapped.

```
dbgcmd

>>> Debug [DebuggableSample <No Position>] >sx
sx
  ab  [  inactive] .... Break on script abort
  eh  [  inactive] .... Break on any thrown exception
  en  [  inactive] .... Break on entry to the script
  uh  [    active] .... Break on unhandled exception
```

Use the **sxe** script debugger command to enable any of the break behaviors. For example to turn on break on entry so that the script will trap into the script debugger as soon as any code within it executes, use this command.

```
dbgcmd

>>> Debug [DebuggableSample <No Position>] >sxe en
sxe en
Event filter 'en' is now active
```

Use the **sxd** script debugger command to disable any of the breakpoint behaviors.

```
dbgcmd

>>> Debug [DebuggableSample 34:5] >sxd en
sxd en
Event filter 'en' is now inactive
```

Stack trace

Use the **k** command to display a stack trace.

```
dbgcmd

>>> Debug [DebuggableSample 34:5] >k
k
## Function
-> [00] throwAndCatch
host.currentProcess)
[01] outer
[02] outermost
Pos      Source Snippet
034:05  (var curProc =
066:05  (var foo = throwAndCatch())
074:05  (var result = outer())
```

Enumerating variables

Use **??** to enumerate the values of JavaScript variables.

```
dbgcmd

>>> Debug [DebuggableSample 34:5] >??someObj
??someObj
someObj       : {...}
__proto__     : {...}
a             : 0x63
b             : {...}
```

Breakpoints

Use the following breakpoint commands to work with additional breakpoints.

bp <bpid>: Set a breakpoint

bd <bpid>: Disable the breakpoint

be <bpid>: Enable the breakpoint

bc <bpid>: Clear the breakpoint

bpc: Set breakpoint on current line

bl: List the breakpoint(s)

Flow control - navigation

Use the following commands to move forward in the script.

p: Step over

t: Step in

g: Continue script

gu: Step out

Frames

Use the following commands to work with frames.

.frame <index>: Switch to frame number <index>

.f+: Switch to next stack frame

.f-: Switch to previous stack frame

Quiting

Use the **.detach** command to detach the JavaScript debugger.

```
dbgcmd  
  
>>> Debug [DebuggableSample 34:5] >.detach  
.detach  
Debugger has been detached from script!
```

Use the **q** command to quit the JavaScript debugger.

```
dbgcmd  
  
>>> Debug [<NONE> ] >q  
q
```

.scriptlist (List Loaded Scripts)

Article • 10/25/2023

The `.scriptlist` command lists the loaded scripts.

```
dbgcmd
  .scriptlist
```

Parameters

None

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The `.scriptlist` command will list any scripts which have been loaded via the `.scriptload` command.

If the TestScript was successfully loaded using `.scriptload`, the `.scriptlist` command would display the name of the loaded script.

```
dbgcmd
  0:000> .scriptlist
  Command Loaded Scripts:
    JavaScript script from 'C:\WinDbg\Scripts\TestScript.js'
```

Requirements

Before using any of the `.script` commands, a scripting provider needs to be loaded.

See also

[JavaScript Debugger Scripting](#)

[.scriptload \(Load Script\)](#)

.scriptload (Load Script)

Article • 10/25/2023

The `.scriptload` command will load and execute the specified script file.

```
dbgcmd
```

```
    .scriptload ScriptFile
```

Parameters

ScriptFile

Specifies the name of the script file to load. *ScriptFile* should include the `.js` file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The `.scriptload` command will load a script and execute a script. The following command shows the successful load of `TestScript.js`.

```
dbgcmd
```

```
0:000> .scriptload C:\WinDbg\Scripts\TestScript.js
JavaScript script successfully loaded from 'C:\WinDbg\Scripts\TestScript.js'
```

If there are any errors in the initial load and execution of the script, the errors will be displayed to console, including the line number and error message.

```
dbgcmd
```

```
0:000:x86> .scriptload C:\WinDbg\Scripts\TestScript.js
0:000> "C:\WinDbg\Scripts\TestScript.js" (line 11 (@ 1)): Error
(0x80004005): Syntax error
Error: Unable to execute JavaScript script 'C:\WinDbg\Scripts\TestScript.js'
```

The `.scriptload` command will execute the following in a JavaScript.

- root code
- `initializeScript` function (if present in the script)

When a script is loaded using the `.scriptload` command, the `initializeScript` function and the root code of the script is executed, the names which are present in the script are bridged into the root namespace of the debugger (dx Debugger) and the script stays resident in memory until it is unloaded and all references to its objects are released.

The script can provide new functions to the debugger's expression evaluator, modify the object model of the debugger, or can act as a visualizers in much the same way that a NatVis visualizer does. For more information about NavVis and the debugger, see [dx \(Display NatVis Expression\)](#).

For more information about working with JavaScript, see [JavaScript Debugger Scripting](#).
For more information about the debugger objects, see [Native Objects in JavaScript Extensions](#).

Requirements

Before using any of the `.script` commands, a scripting provider needs to be loaded. Use the `.scriptproviders` command to confirm that the JavaScript provider is loaded.

```
dbgcmd

0:000> .scriptproviders
Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

See also

[.scriptunload \(Unload Script\)](#)

[JavaScript Debugger Scripting](#)

.scriptproviders (List Script Providers)

Article • 10/25/2023

The `.scriptproviders` command lists the active script providers.

```
dbgcmd
  .scriptproviders
```

Parameters

None

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The `.scriptproviders` command will list all the script languages which are presently understood by the debugger and the extension under which they are registered. Any file ending in ".NatVis" is understood as a NatVis script and any file ending in ".js" is understood as a JavaScript script. Either type of script can be loaded with the `.scriptload` command.

In the example below, the JavaScript and NatVis providers are loaded.

```
dbgcmd
  0:000> .scriptproviders
  Available Script Providers:
    NatVis (extension '.NatVis')
    JavaScript (extension '.js')
```

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded.

See also

[JavaScript Debugger Scripting](#)

[.scriptload \(Load Script\)](#)

.scriptrun (Run Script)

Article • 10/25/2023

The `.scriptrun` command will load and run a JavaScript.

```
dbgcmd
  .scriptrun ScriptFile
```

Parameters

ScriptFile

Specifies the name of the script file to load and execute. *ScriptFile* should include the `.js` file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The `.scriptrun` command will load a script and, execute the following code.

- root
- initializeScript
- invokeScript

A confirmation message is displayed when the code is loaded and executed.

```
dbgcmd
  0:000> .scriptrun C:\WinDbg\Scripts\helloWorld.js
  JavaScript script successfully loaded from 'C:\WinDbg\Scripts\helloWorld.js'
```

```
Hello World!  We are in JavaScript!
```

Any object model manipulations made by the script will stay in place until the script is subsequently unloaded or is run again with different content.

This table summarizes which functions are executed by .scriptload and .scriptrun.

	.scriptload	.scriptrun
root	yes	yes
initializeScript	yes	yes
invokeScript		yes
uninitializeScript		

You can use this code to see which functions are called with the .script run command.

```
dbgcmd

// Root of Script
host.diagnostics.debugLog("****> Code at the very top (root) of the script
is always run \n");

function initializeScript()
{
    // Add code here that you want to run every time the script is loaded.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("****> initializeScript was called \n");
}

function invokeScript()
{
    // Add code here that you want to run every time the script is executed.
    // We will just send a message to indicate that function was called.
    host.diagnostics.debugLog("****> invokeScript was called \n");
}
```

For more information about working with JavaScript, see [JavaScript Debugger Scripting](#). For more information about the debugger objects, see [Native Objects in JavaScript Extensions](#).

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the [.load \(Load Extension DLL\)](#) command to load the JavaScript provider dll.

```
dbgcmd
```

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

See also

[.scriptload \(Load Script\)](#)

[JavaScript Debugger Scripting](#)

.scriptunload (Unload Script)

Article • 10/25/2023

The `.scriptunload` command unloads the specified script.

```
dbgcmd
```

```
    .scriptunload ScriptFile
```

Parameters

ScriptFile

Specifies the name of the script file to unload. *ScriptFile* should include the `.js` file name extension. Absolute or relative paths can be used. Relative paths are relative to the directory that you started the debugger in. File paths containing spaces are not supported.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

The `.scriptunload` command unloads a loaded script. Use the following command syntax to unload a script

```
dbgcmd
```

```
0:000:x86> .scriptunload C:\WinDbg\Scripts\TestScript.js
JavaScript script unloaded from 'C:\WinDbg\Scripts\TestScript.js'
```

If there are outstanding references to objects in a script, the contents of the script will be unlinked but the script may remain in memory until all such references are released.

For more information about working with JavaScript, see [JavaScript Debugger Scripting](#).

For more information about the debugger objects, see [Native Objects in JavaScript Extensions](#).

Requirements

Before using any of the .script commands, a scripting provider needs to be loaded. Use the [.load \(Load Extension DLL\)](#) command to load the JavaScript provider dll.

```
dbgcmd
```

```
0:000> .load C:\ScriptProviders\jsprovider.dll
```

See also

[.scriptload \(Load Script\)](#)

[JavaScript Debugger Scripting](#)

.secure (Activate Secure Mode)

Article • 04/03/2024

The **.secure** command activates or displays the status of Secure Mode.

dbgcmd
.secure 1
.secure

Environment

Secure Mode can only be enabled while the debugger is dormant. Secure Mode applies only to kernel-mode sessions because, by definition, Secure Mode prevents user-mode debugging operations.

[+] [Expand table](#)

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For details, see [Secure Mode](#).

Remarks

To activate Secure Mode, use the command **.secure 1** (or **.secure** followed by any nonzero value).

The command **.secure** will show whether Secure Mode is currently active.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

.send_file (Send File)

Article • 10/25/2023

The **.send_file** command copies files. If you are performing remote debugging through a process server, it sends a file from the smart client's computer to the process server's computer.

dbgcmd

```
.send_file [-f] Source Destination  
.send_file [-f] -s Destination
```

Parameters

-f

Forces file creation. By default, **.send_file** will not overwrite any existing files. If the **-f** switch is used, the destination file will always be created, and any existing file with the same name will be overwritten.

Source

Specifies the full path and filename of the file to be sent. If you are debugging through a process server, this file must be located on the computer where the smart client is running.

Destination

Specifies the directory where the file is to be written. If you are debugging through a process server, this directory name is evaluated on the computer where the process server is running.

-s

Causes the debugger to copy all loaded symbol files.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

This command is particularly useful when you have been performing remote debugging through a process server, but wish to begin debugging locally instead. In this case you can use the .send_file -s command to copy all the symbol files that the debugger has been using to the process server. These symbol files can then be used by a debugger running on the local computer.

.server (Create Debugging Server)

Article • 12/05/2024

The `.server` command starts a debugging server, allowing a remote connection to the current debugging session.

ⓘ Important

There are important security considerations when using remote debugging. For more information, including information on enabling secure mode, see [Security Considerations for Windows Debugging Tools](#).

Recommended connection methods, with some additional security

dbgcmd

```
.server spipe:proto=Protocol,  
{certuser=Cert|machuser=Cert},pipe=PipeName[,hidden][,password=Password]  
.server ssl:proto=Protocol,  
{certuser=Cert|machuser=Cert},port=Socket[,hidden][,password=Password]  
.server ssl:proto=Protocol,  
{certuser=Cert|machuser=Cert},port=Socket,clicon=Client[,password=Password]
```

Unsecure connection methods

dbgcmd

```
.server npipe:pipe=PipeName[,hidden][,password=Password][,IcfEnable]  
.server tcp:port=Socket[,hidden][,password=Password][,ipversion=6]  
[,IcfEnable]  
.server tcp:port=Socket,clicon=Client[,password=Password][,ipversion=6]  
.server com:port=COMPort,baud=BaudRate,channel=COMChannel[,hidden]  
[,password=Password]
```

Parameters

PipeName

When NPIPE or SPIPE protocol is used, *PipeName* is a string that will serve as the name of the pipe. Each pipe name should identify a unique debugging server. If you attempt to reuse a pipe name, you will receive an error message. *PipeName* must not contain spaces or quotation marks. *PipeName* can include a numerical printf-style format code,

such as %x or %d. The debugger will replace this with the process ID of the debugger. A second such code will be replaced with the thread ID of the debugger.

Socket

When TCP or SSL protocol is used, *Socket* is the socket port number.

It is also possible to specify a range of ports separated by a colon. The debugger will check each port in this range to see if it is free. If it finds a free port and no error occurs, the debugging server will be created. The debugging client will have to specify the actual port being used to connect to the server. To determine the actual port, use any of the methods described in [Searching for Debugging Servers](#); when this debugging server is displayed, the port will be followed by two numbers separated by a colon. The first number will be the actual port used; the second can be ignored. For example, if the port was specified as port=51:60, and port 53 was actually used, the search results will show "port=53:60". (If you are using the **cicon** parameter to establish a reverse connection, the debugging client can specify a range of ports in this manner, while the server must specify the actual port used.)

cicon=Client

When TCP or SSL protocol is used and the **cicon** parameter is specified, a *reverse connection* will be opened. This means that the debugging server will try to connect to the debugging client, instead of letting the client initiate the contact. This can be useful if you have a firewall that is preventing a connection in the usual direction. *Client* specifies the network name of the machine on which the debugging client exists or will be created. The two initial backslashes (\\\) are optional.

When **cicon** is used, it is best to start the debugging client before the debugging server is created, although the usual order (server before client) is also permitted. A reverse-connection server will not appear when another debugger displays all active servers.

COMPort

When COM protocol is used, *COMPort* specifies the COM port to be used. The prefix COM is optional (for example, both "com2" and "2" are acceptable).

BaudRate

When COM protocol is used, *BaudRate* specifies the baud rate at which the connection will run. Any baud rate that is supported by the hardware is permitted.

COMChannel

If COM protocol is used, *COMChannel* specifies the COM channel to be used in communicating with the debugging client. This can be any value between 0 and 254, inclusive.

Protocol

If SSL or SPIPE protocol is used, *Protocol* specifies the Secure Channel (S-Channel) protocol. This can be any one of the strings tls1, pct1, ssl2, or ssl3.

Cert

If SSL or SPIPE protocol is used, *Cert* specifies the certificate. This can either be the certificate name or the certificate's thumbprint (the string of hexadecimal digits given by the certificate's snapin). If the syntax **certuser=Cert** is used, the debugger will look up the certificate in the system store (the default store). If the syntax **machuser=Cert** is used, the debugger will look up the certificate in the machine store. The specified certificate must support server authentication.

hidden

Prevents the server from appearing when another debugger displays all active servers.

password=Password

Requires a debugging client to supply the specified password in order to connect to the debugging session. *Password* can be any alphanumeric string, up to twelve characters in length.

ipversion=6

(Debugging Tools for Windows 6.6.07 and earlier only) Forces the debugger to use IP version 6 rather than version 4 when using TCP to connect to the Internet. In Windows Vista and later versions, the debugger attempts to auto-default to IP version 6, making this option unnecessary.

IcfEnable

Causes the debugger to enable the necessary port connections for TCP or named pipe communication when the Internet Connection Firewall is active. By default, the Internet Connection Firewall disables the ports used by these protocols. When **IcfEnable** is used with a TCP connection, the debugger causes Windows to open the port specified by the *Socket* parameter. When **IcfEnable** is used with a named pipe connection, the debugger causes Windows to open the ports used for named pipes (ports 139 and 445). The debugger does not close these ports after the connection terminates.

Environment

[+] [Expand table](#)

Item	Description
Modes	User mode, kernel mode

Item	Description
Targets	Live, crash dump
Platforms	All

Additional Information

For full details on how to start a debugging server, see [Activating a Debugging Server](#). For examples, see [Client and Server Examples](#).

Remarks

This command turns the current debugger into a debugging server. This allows you to start the server after the debugger is already running, whereas the `-server` [command-line option](#) can only be issued when the debugger is started.

This permits a debugging client to connect to the current debugging session. Note that it is possible to start multiple servers using different options, allowing different kinds of debugging clients to join the session.

Important

Using a password with TCP, NPIPE, or COM protocol offers only a small amount of protection, because the password is not encrypted. When you use a password together with a SSL or SPIPE protocol, the password is encrypted. If you want to establish a secure remote session, you must use the SSL or SPIPE protocol.

Feedback

Was this page helpful?

 Yes

 No

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

.servers (List Debugging Servers)

Article • 10/25/2023

The **.servers** command lists all debugging servers that have been established by this debugger.

```
dbgcmd
  .servers
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For full details on debugging servers, see [Remote Debugging Through the Debugger](#).

Remarks

The output of the **.servers** command lists all the debugging servers started by the debugger on which this command is issued. The output is formatted so that it can be used literally as the argument for the **-remote** command-line option or pasted into the WinDbg dialog box.

Each debugging server is identified by a unique ID. This ID can be used as the argument for the [**.endsrv \(End Debugging Server\)**](#) command, if you wish to terminate the debugging server.

The **.servers** command does not list debugging servers started on this computer by different instances of the debugger, nor does it list process servers or KD connection servers.

.setdll (Set Default Extension DLL)

Article • 10/25/2023

The `.setdll` command changes the default extension DLL for the debugger.

```
dbgcmd  
.setdll DLLName  
!DLLName.setdll
```

Parameters

DLLName

The name and path of the extension DLL. If the full path was specified when the DLL was loaded, it needs to be given in full here as well.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details on loading, unloading, and controlling extensions, see [Loading Debugger Extension DLLs](#). For details on executing extension commands, see [Using Debugger Extension Commands](#).

Remarks

The debugger maintains a default extension DLL that is implicitly loaded when the debugger is started. This allows the user to specify an extension command without first having to load an extension DLL. This command allows modification of which DLL is loaded as the default DLL.

When a command is issued, the debugger looks for it in the default extension first. If a match is not found, all other loaded extension DLLs are searched in the order they were loaded.

.shell (Command Shell)

Article • 10/25/2023

The **.shell** command launches a shell process and redirects its output to the debugger, or to a specified file.

dbgcmd

```
.shell [Options] [ShellCommand]
.shell -i InFile [-o OutFile [-e ErrFile]] [Options] ShellCommand
```

Parameters

InFile

Specifies the path and file name of a file to be used for input. If you intend to offer no input after the initial command, you can specify a single hyphen (-) instead of *InFile*, with no space before the hyphen.

OutFile

Specifies the path and file name of a file to be used for standard output. If **-o ****** *OutFile* is omitted, output is sent to the Debugger Command window. If you do not want this output displayed or saved in a file, you can specify a single hyphen (-) instead of *OutFile*, with no space before the hyphen.

ErrFile

Specifies the path and file name of a file to be used for error output. If **-e ErrFile** is omitted, error output is sent to the same place as standard output. If you do not want this output displayed or saved in a file, you can specify a single hyphen (-) instead of *ErrFile*, with no space before the hyphen.

Options Can be any number of the following options:

-ci "Commands"

Processes the specified debugger commands, and then passes their output as an input file to the process being launched. *Commands* can be any number of debugger commands, separated by semicolons, and enclosed in quotation marks.

-x

Causes any process being spawned to be completely detached from the debugger. This allows you to create processes which will continue running even after the debugging session ends.

ShellCommand

Specifies the application command line or Microsoft MS-DOS command to be executed.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For other ways of accessing the command shell, see [Using Shell Commands](#).

Remarks

The **.shell** command is not supported when the output of a user-mode debugger is redirected to the kernel debugger. For more information about redirecting output to the kernel debugger (sometimes called NTSD over KD), see [Controlling the User-Mode Debugger from the Kernel Debugger](#).

The entire line after the **.shell** command will be interpreted as a Windows command (even if it contains a semicolon). This line should not be enclosed in quotation marks. There must be a space between **.shell** and the *ShellCommand* (additional leading spaces are ignored).

The output from the command will appear in the Debugger Command window, unless the **-o **** OutFile** parameter is used.

Issuing a **.shell** command with no parameters will activate the shell and leave it open. All subsequent commands will be interpreted as Windows commands. During this time, the debugger will display messages reading **<.shell process may need input>**, and the WinDbg prompt will be replaced with an **Input>** prompt. Sometimes, a separate Command Prompt window will appear when the debugger leaves the shell open. This window should be ignored; all input and output will be done through the Debugger Command window.

To close this shell and return to the debugger itself, type **exit** or **.shell_quit**. (The **.shell_quit** command is more powerful, because it works even if the shell is frozen.)

This command cannot be used while debugging CSRSS, because new processes cannot be created without CSRSS being active.

You can use the -ci flag to run one or more debugger commands and then pass their output to a shell process. For example, you could pass the output from the [!process 0 7](#) command to a Perl script by using the following command:

```
dbgcmd
```

```
0:000> .shell -ci "!process 0 7" perl.exe parsemyoutput.pl
```

.settings (Set Debug Settings)

Article • 10/25/2023

The `.settings` command sets, modifies, displays, loads and saves settings in the `Debugger.Settings` namespace.

dbgcmd

```
.settings set namespace.setting=value  
.settings set namespace.setting+=value  
.settings save [file path]  
.settings load file path  
.settings list [namespace][-v]  
.settings help
```

Parameters

`.settings set` parameters

namespace.setting=value

Sets or modifies a setting. When specifying file paths use slash escaping, for example `C:\\\\Symbols\\\\`.

Examples:

```
.settings set Display.PreferDMLOutput=false
```

```
.settings set Sources.DisplaySourceLines=true
```

```
.settings set Symbols.Sympath="C:\\\\Symbols\\\\"
```

namespace.setting+=value

Specifies that the new value will be appended to (rather than replace) the previous value.

Example:

```
.settings set Extensions.ExtensionSearchPath+=";C:\\\\MyExtension\\\\"
```

`.setting save` parameters

file path

Saves all of the values in the `Debugger.Settings` namespace to the specified XML file.

none

If a file path is not provided, the settings will be saved to the last file that was loaded or saved to. If a previous file does not exist, a file named config.xml will be created in the directory that the debugger executable was loaded from.

.setting load parameters

file path

Loads all the settings from an XML settings file. Loading settings will change only the settings that are defined in that file. Any previously loaded or changed settings that do not appear in that file will not be modified. This file will be treated as your default save path until the next save or load operation.

.setting list parameters

namespace

List all settings in the given namespace and their values.

-v

The -v flag causes a description of the setting to be displayed.

.setting help parameters

None

Lists all of the settings in the Debugger namespace and their description.

Namespace

Lists all settings in the given namespace and their description.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

On launch, the debugger will load all the settings from config.xml in the directory the debugger executable is in. Throughout your debugging session you can modify settings using the previous settings command (like .sympath or .prefer_dml) or the new .settings

commands. You can use '.settings save' to save your settings to your settings configuration file. You can use the following command to enable AutoSave.

```
.settings set AutoSaveSettings=true
```

When auto save is enabled, the settings in the Debugger.Settings namespace will be automatically saved when exiting the debugger.

Remarks

You can exchange debug xml settings files with others to duplicate their debug settings.

.show_read_failures

Article • 10/25/2023

The `.show_read_failures` command enables or disables the display of read failures.

dbgcmd

```
.show_read_failures /v  
.show_sym_failures /V
```

Parameters

/v

Enables the display of read failures.

/V

Disables the display of read failures.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.show_sym_failures

Article • 10/25/2023

The **.show_sym_failures** command enables or disables the display of symbol lookup failures and type lookup failures.

dbgcmd

```
.show_sym_failures /s  
.show_sym_failures /S  
.show_sym_failures /t  
.show_sym_failures /T
```

Parameters

/s

Enables the display of symbol lookup failures.

/S

Disables the display of symbol lookup failures.

/t

Enables the display of type lookup failures.

/T

Disables the display of type lookup failures.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.sleep (Pause Debugger)

Article • 04/03/2024

The `.sleep` command causes the user-mode debugger to pause and the target computer to become active. This command is only used when you are controlling the user-mode debugger from the kernel debugger.

dbgcmd

```
.sleep milliseconds
```

Parameters

milliseconds

Specifies the length of the pause, in milliseconds.

Environment

 Expand table

Item	Description
Modes	controlling the user-mode debugger from the kernel debugger
Targets	live debugging only
Platforms	all

Additional Information

For details and information about how to wake up a debugger in sleep mode, see [Controlling the User-Mode Debugger from the Kernel Debugger](#).

Remarks

When you are controlling the user-mode debugger from the kernel debugger, and the user-mode debugger prompt is visible in the kernel debugger, this command will activate sleep mode. The kernel debugger, the user-mode debugger, and the target application will all freeze, but the rest of the target computer will become active.

If you use this command in any other scenario, it will simply freeze the debugger for a period of time.

The sleep time is in milliseconds and interpreted according to the default radix, unless a prefix such as **0n** is used. Thus, if the default radix is 16, the following command will cause about 65 seconds of sleep:

```
dbgcmd  
0:000> .sleep 10000
```

Feedback

Was this page helpful?

 Yes

 No

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

.sound_notify (Use Notification Sound)

Article • 04/03/2024

The `.sound_notify` command causes a sound to be played when WinDbg enters the wait-for-command state.

```
dbgcmd  
.sound_notify /ed  
.sound_notify /ef File  
.sound_notify /d
```

Parameters

/ed

Causes the default Windows alert sound to be played when WinDbg enters the wait-for-command state.

/ef **** *File*

Causes the sound contained in the specified file to be played when WinDbg enters the wait-for-command state.

/d

Disables the playing of sounds.

Environment

This command is available only in WinDbg and cannot be used in script files.

[] Expand table

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Feedback

Was this page helpful?

 Yes

 No

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

.srcfix, .lsrcfix (Use Source Server)

Article • 10/25/2023

The **.srcfix** and **.lsrcfix** commands automatically set the source path to indicate that a source server will be used.

dbgcmd

```
.srcfix[+] [Paths]  
.lsrcfix[+] [Paths]
```

Parameters

+

Causes the existing source path to be preserved, and ;**srv*** to be appended to the end. If the + is not used, the existing source path is replaced.

Paths

Specifies one or more additional paths to append to the end of the new source path.

Environment

The **.srcfix** command is available on all debuggers. The **.lsrcfix** command is available only in WinDbg and cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more information on setting the local source path for a remote client, see [WinDbg Command-Line Options](#). For details about **SrcSrv**, see [Using a Source Server](#). For details on the source path and the local source path, see [Source Path](#). For more information about commands that can be used while performing remote debugging through the debugger, see [Controlling a Remote Debugging Session](#).

Remarks

When you add `srv*` to the source path, the debugger uses [SrcSrv](#) to retrieve source files from locations specified in the target modules' symbol files. Using `srv*` in the source path is fundamentally different from using `srv*` in the symbol path. In the symbol path, you can specify a symbol server location along with the `srv*` (for example, `.sympath SRV*https://msdl.microsoft.com/download/symbols`). In the source path, `srv*` stands alone, separated from all other elements by semicolons.

When this command is issued from a debugging client, `.srcfix` sets the source path to use a source server on the debugging server, while `.lsrcfix` does the same on the local machine.

These commands are the same as the [.srcpath \(Set Source Path\)](#) and [.lsrcpath \(Set Local Source Path\)](#) commands followed by the `srv*` source path element. Thus, the following two commands are equivalent:

```
dbgcmd  
.srcfix[+] [Paths]  
.srcpath[+] srv*[;Paths]
```

Similarly, the following two commands are equivalent:

```
dbgcmd  
.lsrcfix[+] [Paths]  
.lsrcpath[+] srv*[;Paths]
```

.srcnoisy (Noisy Source Loading)

Article • 10/25/2023

The `.srcnoisy` command controls the verbosity level for source file loading.

```
dbgcmd  
.srcnoisy [Options]
```

Parameters

Options Can be any one of the following options:

0

Disables the display of extra messages.

1

Displays information about the progress of source file loading and unloading.

2

Displays information about the progress of symbol file loading and unloading.

3

Displays all information displayed by options 1 and 2.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

With no parameters, `.srcnoisy` will display the current status of noisy source loading.

Noisy source loading should not be confused with noisy symbol loading -- that is controlled by the [!sym noisy](#) extension and by other means of controlling the [SYMOPT_DEBUG](#) setting.

.srcpath, .lsrcpath (Set Source Path)

Article • 10/25/2023

The **.srcpath** and **.lsrcpath** commands set or display the source file search path.

dbgcmd

```
.srcpath[+] [Directory [; ...]]  
.lsrcpath[+] [Directory [; ...]]
```

Parameters

+

Specifies that the new directories will be appended to (rather than replacing) the previous source file search path.

Directory

Specifies one or more directories to put in the search path. If *Directory* is not specified, the current path is displayed. Separate multiple directories with semicolons.

Environment

The **.srcpath** command is available on all debuggers. The **.lsrcpath** command is available only in WinDbg and cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details and other ways to change this path, see [Source Path](#). For more information about commands that can be used while performing remote debugging through the debugger, see [Controlling a Remote Debugging Session](#).

 Note

The source path in WinDbg supports file retrieval using Source Link 1.0, and starting version 1.2104, file retrieval using Source Link 2.0 or DebugInfoD servers. For more information on source path syntax, see [Source Code Extended Access](#).

Remarks

If you include `srv*` in your source path, the debugger uses [SrcSrv](#) to retrieve source files from locations specified in the target modules' symbol files. For more information about using `srv*` in a source path, see [Using a Source Server](#) and [.srcfix](#).

When this command is issued from a debugging client, `.srcpath` sets the source path on the debugging server, while `.lsrcpath` sets the source path on the local machine.

.step_filter (Set Step Filter)

Article • 10/25/2023

The `.step_filter` command creates a list of functions that are skipped (stepped over) when tracing. This allows you to trace through code and skip only certain functions. It can also be used in source mode to control stepping when there are multiple function calls on one line.

```
dbgcmd  
.step_filter "FilterList"  
.step_filter /c  
.step_filter
```

Parameters

"*FilterList*"

Specifies the symbols associated with functions to be stepped over. *FilterList* can contain any number of text patterns separated by semicolons. Each of these patterns may contain a variety of wildcards and specifiers; see [String Wildcard Syntax](#) for details. A function whose symbol matches at least one of these patterns will be stepped over during tracing. Each time "*FilterList*" is used, any previous filter list is discarded and completely replaced with the new list.

/c

Clears the filter list.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Without any parameters, `.step_filter` displays the current filter list.

Typically, a trace command (for example, `t` or the windbg `debug | step into`) traces into a function call. However, if the symbol associated with the function being called matches a pattern specified by *FilterList*, the function will be stepped over -- as if a step command (for example, `p`) had been used.

If the instruction pointer is located within code that is listed in the filter list, any trace or step commands will step out of this function, like the `gu` command or the WinDbg Step Out button. Of course, this filter would prevent such code from having been traced into in the first place, so this will only happen if you have changed the filter or hit a breakpoint.

For example, the following command will cause trace commands to skip over all CRT calls:

```
dbgcmd  
.step_filter "msvcrt!"*
```

The `.step_filter` command is most useful when you are debugging in source mode, because there can be multiple function calls on a single source line. The `p` and `t` commands cannot be used to separate these function calls.

For example, in the following line, the `t` command will step into both `GetTickCount` and `printf`, while the `p` command will step over both function calls:

```
dbgcmd  
printf( "%x\n", GetTickCount() );
```

The `.step_filter` command allows you to filter out one of these calls while still tracing into the other.

Because the functions are identified by symbol, a single filter can include an entire module. This lets you filter out framework functions -- for example, Microsoft Foundation Classes (MFC) or Active Template Library (ATL) calls.

When debugging in assembly mode, each call is on a different line, so you can choose whether to step or trace line-by-line. So `.step_filter` is not very useful in assembly mode.

.suspend_ui (Suspend WinDbg Interface)

Article • 04/03/2024

The **.suspend_ui** command suspends the refresh of WinDbg debugging information windows.

```
dbgcmd  
.suspend_ui 0  
.suspend_ui 1  
.suspend_ui
```

Parameters

0

Suspends the refresh of WinDbg debugging information windows.

1

Enables the refresh of WinDbg debugging information windows.

Environment

This command is available only in WinDbg and cannot be used in script files.

 Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Remarks

Without any parameters, **.suspend_ui** displays whether debugging information windows are currently suspended.

By default, debugging information windows are refreshed every time the information they display changes.

Suspending the refresh of these windows can speed up WinDbg during a sequence of quick operations -- for example, when tracing or stepping many times in quick succession.

Feedback

Was this page helpful?



Yes



No

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

.symfix (Set Symbol Store Path)

Article • 10/25/2023

The `.symfix` command automatically sets the symbol path to point to the Microsoft symbol store.

dbgcmd

```
.symfix[+] [LocalSymbolCache]
```

Parameters

`+`

Causes the Microsoft symbol store path to be appended to the existing symbol path. If this is not included, the existing symbol path is replaced.

LocalSymbolCache

Specifies the directory to be used as a local symbol cache. If this directory does not exist, it will be created when the symbol server begins copying files. If *LocalSymbolCache* is omitted, the sym subdirectory of the debugger installation directory will be used.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details, see [Using Symbol Servers and Symbol Stores](#).

Remarks

The following example shows how to use `.symfix` to set a new symbol path that points to the Microsoft symbol store.

```
dbgcmd
```

```
3: kd> .symfix c:\myCache
3: kd> .sympath
Symbol search path is: srv*
Expanded Symbol search path is:
cache*c:\myCache;SRV*https://msdl.microsoft.com/download/symbols
```

The following example shows how to use `.symfix+` to append the existing symbol path with a path that points to the Microsoft symbol store.

```
dbgcmd
```

```
3: kd> .sympath
Symbol search path is: c:\someSymbols
Expanded Symbol search path is: c:\somesymbols
3: kd> .symfix+ c:\myCache
3: kd> .sympath
Symbol search path is: c:\someSymbols;srv*
Expanded Symbol search path is:
c:\somesymbols;cache*c:\myCache;SRV*https://msdl.microsoft.com/download/symbols
```

.symopt (Set Symbol Options)

Article • 10/25/2023

The `.symopt` command sets or displays the symbol options.

dbgcmd

```
.symopt+ Flags  
.symopt- Flags  
.symopt
```

Parameters

+

Causes the symbol options specified by *Flags* to be set. If `.symopt` is used with *Flags* but no plus or minus sign, a plus sign is assumed.

-

Causes the symbol options specified by *Flags* to be cleared.

Flags

Specifies the symbol options to be changed. *Flags* must be the sum of the bit flags of these symbol options.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For a list and description of each symbol option, its bit flag, and other methods of setting and clearing these options, see [Setting Symbol Options](#).

Remarks

Without any arguments, **.symopt** displays the current symbol options.

.sympath (Set Symbol Path)

Article • 10/25/2023

The **.sympath** command sets or alters the symbol path. The symbol path specifies locations where the debugger looks for symbol files.

dbgcmd

```
.sympath[+] [Path [; ...]]
```

Parameters

+

Specifies that the new locations will be appended to (rather than replace) the previous symbol search path.

Path

A fully qualified path or a list of fully qualified paths. Multiple paths are separated by semicolons. If *Path* is omitted, the current symbol path is displayed.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For details and other ways to change this path, see [Symbol Path](#).

Remarks

New symbol information will not be loaded when the symbol path is changed. You can use the [.reload \(Reload Module\)](#) command to reload symbols.

.thread (Set Register Context)

Article • 10/25/2023

The `.thread` command specifies which thread will be used for the register context.

```
dbgcmd
```

```
.thread [/p [/r] ] [/P] [/w] [Thread]
```

Parameters

/p

(Live debugging only) If this option is included and *Thread* is nonzero, all transition page table entries (PTEs) for the process owning this thread will be automatically translated into physical addresses before access. This may cause slowdowns, because the debugger will have to look up the physical addresses for all the memory used by this process, and a significant amount of data may need to be transferred across the debug cable. (This behavior is the same as that of [.cache forcedecodeuser](#).)

If the `/p` option is included and *Thread* is zero or omitted, this translation will be disabled. (This behavior is the same as that of [.cache noforcedecodeuser](#).)

/r

(Live debugging only) If the `/r` option is included along with the `/p` option, user-mode symbols for the process owning this thread will be reloaded after the process and register contexts have been set. (This behavior is the same as that of [.reload /user](#).)

/P

(Live debugging only) If this option is included and *Thread* is nonzero, all transition page table entries (PTEs) will be automatically translated into physical addresses before access. Unlike the `/p` option, this translates the PTEs for all user-mode and kernel-mode processes, not only the process owning this thread. This may cause slowdowns, because the debugger will have to look up the physical addresses for all memory in use, and a huge amount of data may need to be transferred across the debug cable. (This behavior is the same as that of [.cache forcedecodeptes](#).)

/w

(64-bit kernel debugging only) Changes the active context for the thread to the WOW64 32-bit context. The thread specified must be running in a process that has a WOW64 state.

Thread

The address of the thread. If this is omitted or zero, the thread context is reset to the current thread.

Environment

Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the register context and other context settings, see [Changing Contexts](#).

Remarks

Generally, when you are doing kernel debugging, the only visible registers are the ones associated with the current thread.

The **.thread** command instructs the kernel debugger to use the specified thread as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context persists until you allow the target to execute or use another register context command (**.thread**, **.cxr**, or **.trap**). See [Register Context](#) for full details.

The **/w** option can only be used in 64-bit kernel debugging sessions on a thread running in a process that has a WOW64 state. The context retrieved will be the last context remembered by WOW64; this is usually the last user-mode code executed by *Thread*. This option can only be used if the target is in native machine mode. For example, if the target is running on a 64-bit machine that is emulating an x86-based processor using WOW64, this option cannot be used. Using the **/w** option will cause the machine mode to switch automatically to an x86-based processor.

This command does not actually change the current thread. In other words, extensions such as **!thread** and **!teb** will still default to the current thread if no arguments are used with them.

Here is an example. Use the **!process** extension to find the address of the desired thread. (In this case, **!process 0 0** is used to list all processes, then **!process** is used a

second time to list all the threads for the desired process.)

```
dbgcmd

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fe5039e0 SessionId: 0 Cid: 0008 Peb: 00000000 ParentCid: 0000
    DirBase: 00030000 ObjectTable: fe529a88 TableSize: 145.
    Image: System

.....
PROCESS ffaa5280 SessionId: 0 Cid: 0120 Peb: 7ffdf000 ParentCid: 01e0
    DirBase: 03b70000 ObjectTable: ffaa4e48 TableSize: 23.
    Image: winmine.exe

kd> !process ffaa5280
PROCESS ffaa5280 SessionId: 0 Cid: 0120 Peb: 7ffdf000 ParentCid: 01e0
    DirBase: 03b70000 ObjectTable: ffaa4e48 TableSize: 23.
    Image: winmine.exe
    VadRoot ffaf6e48 Clone 0 Private 50. Modified 0. Locked 0.
    DeviceMap fe502e88
    Token e1b55d70

.....
THREAD ffaa43a0 Cid 120.3a4 Teb: 7ffde000 Win32Thread: e1b4fea8
WAIT: (WrUserRequest) UserMode Non-Alertable
    ffadc6a0 SynchronizationEvent
    Not impersonating
    Owning Process ffaa5280
    WaitTime (seconds) 24323
    Context Switch Count 494 LargeStack

.....
```

Now use the `.thread` command with the address of the desired thread. This sets the register context and enables you to examine the important registers and the call stack for this thread.

```
dbgcmd

kd> .thread ffaa43a0
Using context of thread ffaa43a0

kd> r
Last set context:
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000
edi=00000000
eip=80403a0d esp=fd581c2c ebp=fd581c60 iopl=0 nv up di pl nz na pe
nc
cs=0000 ss=0000 ds=0000 es=0000 fs=0000 gs=0000
```

```
efl=00000000  
0000:3a0d ??  
  
kd> k  
*** Stack trace for last set context - .thread resets it  
ChildEBP RetAddr  
fd581c38 8042d61c ntoskrnl!KiSwapThread+0xc5  
00001c60 00000000 ntoskrnl!KeWaitForSingleObject+0x1a1
```

.time (Display System Time)

Article • 10/25/2023

The `.time` command displays information about the system time variables.

```
dbgcmd
```

```
  .time [-h Hours]
```

Parameters

-h ** Hours**

Specifies the offset from Greenwich Mean Time, in hours. A negative value of *Hours* must be enclosed in parentheses.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The system time variables control performance counter behavior.

Here is an example in kernel mode:

```
dbgcmd
```

```
kd> .time
Debug session time: Wed Jan 31 14:47:08 2001
System Uptime: 0 days 2:53:56
```

Here is an example in user mode:

```
dbgcmd
```

```
0:000> .time
Debug session time: Mon Apr 07 19:10:50 2003
System Uptime: 4 days 4:53:56.461
Process Uptime: 0 days 0:00:08.750
    Kernel time: 0 days 0:00:00.015
    User time: 0 days 0:00:00.015
```

.tlist (List Process IDs)

Article • 10/25/2023

The `.tlist` command lists all processes running on the system.

dbgcmd

```
.tlist [Options][FileNamePattern]
```

Parameters

Options Can be any number of the following options:

`-v`

Causes the display to be verbose. This includes the session number, the process user name, and the command-line used to start the process.

`-c`

Limits the display to just the current process.

FileNamePattern

Specifies the file name to be displayed, or a pattern that the file name of the process must match. Only those processes whose file names match this pattern will be displayed. *FileNamePattern* may contain a variety of wildcards and specifiers; see [String Wildcard Syntax](#) for details. This match is made only against the actual file name, not the path.

Environment

Item	Description
Modes	user mode only
Targets	live debugging only
Platforms	all

Additional Information

For other methods of displaying processes, see [Finding the Process ID](#).

Remarks

Each process ID is displayed with an **0n** prefix, to emphasize that the PID is a decimal number.

.trap (Display Trap Frame)

Article • 04/03/2024

The `.trap` command displays the trap frame register state and also sets the register context.

```
dbgcmd
```

```
.trap [Address]
```

Parameters

Address

Hexadecimal address of the trap frame on the target system. Omitting the address does not display any trap frame information, but it does reset the register context.

Environment

 Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	all

Additional Information

For more information about the register context and other context settings, see [Changing Contexts](#).

Remarks

The `.trap` command displays the important registers for the specified trap frame.

This command also instructs the kernel debugger to use the specified context record as the register context. After this command is executed, the debugger will have access to the most important registers and the stack trace for this thread. This register context

persists until you allow the target to execute or use another register context command ([.thread](#), [.cxr](#), or [.trap](#)). See [Register Context](#) for full details.

This extension is commonly used when debugging bug check 0xA and 0x7F. For details and an example, see [Bug Check 0xA](#) (IRQL_NOT_LESS_OR_EQUAL).

Feedback

Was this page helpful?



Yes



No

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

.tss (Display Task State Segment)

Article • 04/03/2024

The `.tss` command displays a formatted view of the saved Task State Segment (TSS) information for the current processor.

```
dbgcmd
  .tss [Address]
```

Parameters

Address

Address of the TSS.

Environment

[] Expand table

Item	Description
Modes	kernel mode only
Targets	live, crash dump
Platforms	x86 only

Remarks

The address of the TSS can be found by examining the output of the [!pcr](#) extension.

Feedback

Was this page helpful?

 Yes

 No

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

.ttime (Display Thread Times)

Article • 04/03/2024

The **.ttime** command displays the running times for a thread.

```
dbgcmd
.ttime
```

Environment

[+] Expand table

Item	Description
Modes	user mode only
Targets	live, crash dump
Platforms	x86 only

Remarks

This command only works in user mode. In kernel mode you should use [!thread](#) instead. This command works with user-mode minidumps as long as they were created with the `/mt` or `/ma` options; see [User-Mode Dump Files](#) for details.

The **.ttime** command shows the creation time of the thread, as well as the amount of time it has been running in kernel mode and in user mode.

Here is an example:

```
dbgcmd
0:000> .ttime
Created: Sat Jun 28 17:58:42 2003
Kernel: 0 days 0:00:00.131
User:    0 days 0:00:02.109
```

Feedback

Was this page helpful?

 Yes

 No

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

.typeopt (Set Type Options)

Article • 10/25/2023

The `.typeopt` command sets or displays the type options.

```
dbgcmd  
.typeopt +Flags  
.typeopt -Flags  
.typeopt +FlagName  
.typeopt -FlagName  
.typeopt
```

Parameters

`+`

Causes the type option(s) specified by *Flags* or *FlagName* to be set.

`-`

Causes the type option(s) specified by *Flags* or *FlagName* to be cleared.

Flags

Specifies the type options to be changed. *Flags* can be a sum of any of the following values (there is no default):

0x1

Displays unsigned 16-bit integer values in all Watch windows and the Locals window as having UNICODE data type.

0x2

Displays signed 32-bit integer values in all Watch windows and the Locals window as unsigned integers in the default radix.

0x4

Displays signed integers of all sizes in all Watch windows and the Locals window as unsigned values in the default radix.

0x8

Causes the debugger to choose the matching symbol with the largest size when the Locals window or Watch window references a symbol by name but there is more than one symbol that matches this name. The size of a symbol is defined as follows: if the

symbol is the name of a function, its size is the size of the function in memory. Otherwise, the size of the symbol is the size of the data type that it represents.

FlagName

Specifies the type options to be changed. *FlagName* can be any one of the following strings (there is no default):

uni

Displays unsigned 16-bit integer values in all Watch windows and the Locals window as having UNICODE data type. (This has the same effect as **0x1**.)

longst

Displays signed 32-bit integer values in all Watch windows and the Locals window as unsigned integers in the default radix. (This has the same effect as **0x2**.)

radix

Displays signed integers of all sizes in all Watch windows and the Locals window as unsigned values in the default radix. (This has the same effect as **0x4**.)

size

Causes the debugger to choose the matching symbol with the largest size when the Locals window or Watch window references a symbol by name but there is more than one symbol that matches this name. The size of a symbol is defined as follows: if the symbol is the name of a function, its size is the size of the function in memory. Otherwise, the size of the symbol is the size of the data type that it represents. (This has the same effect as **0x8**.)

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Without any arguments, **.typeopt** displays the current symbol options.

To change the default radix, use the [n \(Set Number Base\)](#) command.

.unload (Unload Extension DLL)

Article • 10/25/2023

The `.unload` command unloads an extension DLL from the debugger.

```
dbgcmd
```

```
.unload DLLName  
!DLLName.unload
```

Parameters

DLLName

Specifies the file name of the debugger extension DLL to be unloaded. If the full path was specified when the DLL was loaded, it needs to be given in full here as well. If *DLLName* is omitted, the current extension DLL is unloaded.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more details on loading, unloading, and controlling extensions, see [Loading Debugger Extension DLLs](#).

Remarks

This command is useful when testing an extension you are creating. When the extension is recompiled, you must unload and then load the new DLL.

.unloadall (Unload All Extension DLLs)

Article • 10/25/2023

The **.unloadall** command unloads all extension DLLs from the debugger on the host system.

```
dbgcmd
```

```
.unloadall
```

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Additional Information

For more details on loading, unloading, and controlling extensions, see [Loading Debugger Extension DLLs](#).

.urestart (Unregister for Restart)

Article • 10/25/2023

The **.urestart** command unregisters the debugging session for restart in case of a reboot or an application failure.

```
dbgcmd
```

```
    .urestart
```

Remarks

This command does not work for elevated debugger sessions.

See also

[.rrestart](#)

.wake (Wake Debugger)

Article • 04/03/2024

The **.wake** command causes sleep mode to end. This command is used only when you are controlling the user-mode debugger from the kernel debugger.

```
dbgcmd
```

```
.wake PID
```

Parameters

PID

The process ID of the user-mode debugger.

Environment

[] [Expand table](#)

Item	Description
Modes	controlling the user-mode debugger from the kernel debugger
Targets	live debugging only
Platforms	all

Additional Information

For more details, see [Controlling the User-Mode Debugger from the Kernel Debugger](#).

For information about how to find the process ID of the debugger, see [Finding the Process ID](#).

Remarks

When you are controlling the user-mode debugger from the kernel debugger and the system is in sleep mode, this command can be used to wake up the debugger before the sleep timer runs out.

This command is not issued in the user-mode debugger on the target machine, nor in the kernel debugger on the host machine. It must be issued from a third debugger (KD, CDB, or NTSD) running on the target machine.

This debugger can be started expressly for this purpose, or can be another debugger that happens to be running. However, if there is no other debugger already running, it is easier just to use CDB with the [-wake command-line option](#).

Feedback

Was this page helpful?

 Yes

 No

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

.write_cmd_hist (Write Command History)

Article • 10/25/2023

The **.write_cmd_hist** command writes the entire history of the Debugger Command window to a file.

```
dbgcmd  
.write_cmd_hist Filename
```

Parameters

Filename

Specifies the path and filename of the file that will be created.

Environment

This command is available only in WinDbg and cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

.writemem (Write Memory to File)

Article • 10/25/2023

The **.writemem** command writes a section of memory to a file.

```
dbgcmd
```

```
.writemem FileName Range
```

Parameters

FileName

Specifies the name of the file to be created. You can specify a full path and file name, or just the file name. If the file name contains spaces, *FileName* should be enclosed in quotation marks. If no path is specified, the current directory is used.

Range

Specifies the memory range to be written to the file. For syntax details, see [Address and Address Range Syntax](#).

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

The memory is copied literally to the file. It is not parsed in any way.

The **.writemem** command is the opposite of the [.readmem \(Read Memory from File\)](#) command.

.wtitle (Set Window Title)

Article • 10/25/2023

The **.wtitle** command sets the title in the main WinDbg window or in the NTSD, CDB, or KD window.

```
dbgcmd  
.wtitle Title
```

Parameters

Title

The title to use for the window.

Environment

This command cannot be used in script files.

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

For CDB, NTSD, or KD, if the **.wtitle** command has not been used, the window title matches the command line used to launch the debugger.

For WinDbg, if **.wtitle** has not been used, the main window title includes the name of the target. If a debugging server is active, its connection string is displayed as well. If multiple debugging servers are active, only the most recent one is displayed.

When **.wtitle** is used, *Title* replaces all this information. Even if a debugging server is started later, *Title* will not change.

The WinDbg version number is always displayed in the window title bar, regardless of whether this command is used.

General Extensions

Article • 10/25/2023

This section describes extension commands that are frequently used during both user-mode and kernel-mode debugging.

The debugger automatically loads the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions that are being used. For more information about the default module search order, see [Using Debugger Extension Commands](#). For more information about how to load extension modules, see [Loading Debugger Extension DLLs](#).

Each extension command reference topics lists the DLLs that expose that command. Use the following rules to determine the proper directory to load this extension DLL from:

- If your target computer is running Microsoft Windows XP or a later version of Windows, use `winxp\kdexts.dll`, `winxp\ntsdexts.dll`, `winxp\exts.dll`, `winext\ext.dll`, or `dbghelp.dll`.

!acl

Article • 10/25/2023

The **!acl** extension formats and displays the contents of an access control list (ACL).

Syntax

```
dbgcmd  
!acl Address [Flags]
```

Parameters

Address

Specifies the hexadecimal address of the ACL.

Flags

Displays the friendly name of the ACL, if the value of *Flags* is 1. This friendly name includes the security identifier (SID) type and the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For more information about access control lists, see [!sid](#), [!sd](#), and [Determining the ACL of an Object](#). Also, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The following example shows the **!acl** extension.

```
Console  
  
kd> !acl e1bf35d4 1  
ACL is:  
ACL is: ->AclRevision: 0x2  
ACL is: ->Sbz1 : 0x0
```

```
ACL is: ->AclSize      : 0x40
ACL is: ->AceCount     : 0x2
ACL is: ->Sbz2         : 0x0
ACL is: ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
ACL is: ->Ace[0]: ->AceFlags: 0x0
ACL is: ->Ace[0]: ->AceSize: 0x24
ACL is: ->Ace[0]: ->Mask : 0x10000000
ACL is: ->Ace[0]: ->SID: S-1-5-21-518066528-515770016-299552555-2981724
(User: MYDOMAIN\myuser)

ACL is: ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
ACL is: ->Ace[1]: ->AceFlags: 0x0
ACL is: ->Ace[1]: ->AceSize: 0x14
ACL is: ->Ace[1]: ->Mask : 0x10000000
ACL is: ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
```

!address

Article • 10/25/2023

The **!address** extension displays information about the memory that the target process or target computer uses.

User-Mode

```
dbgcmd
```

```
!address Address
!address -summary
!address [-f:F1,F2,...] {[-o:{csv | tsv | 1}] | [-c:"Command"]}
!address -? | -help
```

Kernel-Mode

```
dbgcmd
```

```
!address Address
!address
```

Parameters

Address

Displays only the region of the address space that contains *Address*.

-summary

Displays only summary information.

-f:*F1, F2, ...*

Displays only the regions specified by the filters *F1, F2*, and so on.

The following filter values specify memory regions by the way that the target process is using them.

Filter value	Memory regions displayed
VAR	Busy regions. These regions include all virtual allocation blocks, the SBH heap, memory from custom allocators, and all other regions of the address space that fall into no other classification.

Filter value	Memory regions displayed
Free	Free memory. This includes all memory that has not been reserved.
Image	Memory that is mapped to a file that is part of an executable image.
Stack	Memory used for thread stacks.
Teb	Memory used for thread environment blocks (TEBs).
Peb	Memory used for the process environment block (PEB).
Heap	Memory used for heaps.
PageHeap	The memory region used for the full-page heap.
CSR	CSR shared memory.
Actx	Memory used for activation context data.
NLS	Memory used for National Language Support (NLS) tables.
FileMap	Memory used for memory-mapped files. This filter is applicable only during live debugging.

The following filter values specify memory regions by the memory type.

Filter value	Memory regions displayed
MEM_IMAGE	Memory that is mapped to a file that is part of an executable image.
MEM_MAPPED	Memory that is mapped to a file that is not part of an executable image. This includes memory that is mapped to the paging file.
MEM_PRIVATE	Private memory. This memory is not shared by any other process, and it is not mapped to any file.

The following filter values specify memory regions by the state of the memory.

Filter value	Memory regions displayed
MEM_COMMIT	Committed memory.

Filter value	Memory regions displayed
MEM_FREE	Free memory. This includes all memory that has not been reserved.
MEM_RESERVE	Reserved memory.

The following filter values specify memory regions by the protection applied to the memory.

Filter value	Memory regions displayed
PAGE_NOACCESS	Memory that cannot be accessed.
PAGE_READONLY	Memory that is readable, but not writable and not executable.
PAGE_READWRITE	Memory that is readable and writable, but not executable.
PAGE_WRITECOPY	Memory that has copy-on-write behavior.
PAGE_EXECUTE	Memory that is executable, but not readable and not writeable.
PAGE_EXECUTE_READ	Memory that is executable and readable, but not writable.
PAGE_EXECUTE_READWRITE	Memory that is executable, readable, and writable.
PAGE_EXECUTE_WRITECOPY	Memory that is executable and has copy-on-write behavior.
PAGE_GUARD	Memory that acts as a guard page.
PAGE_NOCACHE	Memory that is not cached.
PAGE_WRITECOMBINE	Memory that has write-combine access enabled.

-o:{csv | tsv | 1}

Displays the output according to one of the following options.

Option	Output format
csv	Displays the output as comma-separated values.
tsv	Displays the output as tab-separated values.

Option	Output format
1	Displays the output in bare format. This format works well when !address is used as input to .foreach .

-c:"Command"

Executes a custom command for each memory region. You can use the following placeholders in your command to represent output fields of the !address extension.

Placeholder	Output field
%1	Base address
%2	End address + 1
%3	Region size
%4	Type
%5	State
%6	Protection
%7	Usage

For example, `!address -f:Heap -c:".echo %1 %3 %5"` displays the base address, size, and state for each memory region of type **Heap**.

Quotes in the command must be preceded by a back slash (\"). For example, `!address -f:Heap -c:"s -a %1 %2 \"pad\""` searches each memory region of type **Heap** for the string "pad".

Multiple commands separated by semicolons are not supported.

-?

Displays minimal Help text for this extension in the [Debugger Command window](#).

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to display and search memory, see [Reading and Writing Memory](#). For additional extensions that display memory properties, see [!vm](#) (kernel mode) and [!vprot](#) (user mode).

Remarks

Without any parameters, the **!address** extension displays information about the whole address space. The **!address -summary** command shows only the summary.

In kernel mode, this extension searches only kernel memory, even if you used [.process \(Set Process Context\)](#) to specify a given process' virtual address space. In user mode, the **!address** extension always refers to the memory that the target process owns.

In user mode, **!address Address** shows the characteristics of the region that the specified address belongs to. Without parameters, **!address** shows the characteristics of all memory regions. These characteristics include the memory usage, memory type, memory state, and memory protection. For more information about the meaning of this information, see the earlier tables in the description of the **-f** parameter.

The following example uses **!address** to retrieve information about a region of memory that is mapped to kernel32.dll.

```
Console

0:000> !address 75831234
Usage:           Image
Base Address:    75831000
End Address:    758f6000
Region Size:    000c5000
Type:            01000000MEM_IMAGE
State:           00001000MEM_COMMIT
Protect:         00000020PAGE_EXECUTE_READ
More info:       !mv m kernel32
More info:       !lmi kernel32
More info:       ln 0x75831234
```

This example uses an **Address** value of 0x75831234. The display shows that this address is in a memory region that begins with the address 0x75831000 and ends with the address 0x758f6000. The region has usage **Image**, type **MEM_IMAGE**, state **MEM_COMMIT**, and protection **PAGE_EXECUTE_READ**. (For more information about the meaning of these values, see the earlier tables.) The display also lists three other debugger commands that you can use to get more information about this memory address.

If you are starting with an address and trying to determine information about it, the usage information is frequently the most valuable. After you know the usage, you can use additional extensions to learn more about this memory. For example, if the usage is **Heap**, you can use the **!heap** extension to learn more.

The following example uses the **s (Search Memory)** command to search each memory region of type **Image** for the wide-character string "Note".

```
Console

!address /f:Image /c:"s -u %1 %2 \"Note\)"

*** Executing: s -u 0xab0000 0xab1000 "Note"
*** Executing: s -u 0xab1000 0xabc000 "Note"
00ab2936 004e 006f 0074 0065 0070 0061 0064 0000 N.o.t.e.p.a.d...
00ab2f86 004e 006f 0074 0065 0070 0061 0064 005c N.o.t.e.p.a.d.\.
00ab32e4 004e 006f 0074 0065 0070 0061 0064 0000 N.o.t.e.p.a.d...
*** Executing: s -u 0xabc000 0abd000 "Note"
. . .
```

In kernel mode, the output of **!address** is similar to the user mode output but contains less information. The following example shows the kernel mode output.

```
Console

kd> !address
804de000 - 00235000
Usage      KernelSpaceUsageImage
ImageName   ntoskrnl.exe

80c00000 - 001e1000
Usage      KernelSpaceUsagePFNDatabase

....
f85b0000 - 00004000
Usage      KernelSpaceUsageKernelStack
KernelStack 817b4da0 : 324.368

f880d000 - 073d3000
Usage      KernelSpaceUsageNonPagedPoolExpansion
```

The meaning of "usage" is the same as in user mode. "ImageName" indicates the module that is associated with this address. "KernelStack" shows the address of this thread's ETHREAD block (0x817B4DA0), the process ID (0x324), and the thread ID (0x368).

!analyze (WinDbg)

Article • 10/25/2023

The **!analyze** extension displays information about the current exception or bug check.

User-Mode

dbgcmd

```
!analyze [-v[0..99]] [-f | -hang]
!analyze [-v[0..99]] -xml [-xmi] [-xcs] [-xmf OutputXmlFile]
!analyze -c [-load KnownIssuesFile | -unload | -help ]
```

Kernel-Mode

dbgcmd

```
!analyze [-v[0..99]] [-f | -hang]
!analyze -show BugCheckCode [BugParameters]
!analyze [-v[0..99]] -xml [-xmi] [-xcs] [-xmf OutputXmlFile]
!analyze -c [-load KnownIssuesFile | -unload | -help ]
```

General parameters

-v[0..99]

Displays verbose output. You can display more information by specifying a number from 0 to 99. If you don't specify a number, the default value is 1. You can also specify Very Verbose (-vv) to display all available information.

For user mode, **-v6** displays what has been discovered globally and on each thread.

-f

Generates the **!analyze** exception output. Use this parameter to see an exception analysis even when the debugger doesn't detect an exception.

-hang

Generates **!analyze** hung-application output. Use this parameter when the target has experienced a bug check or exception. However, an analysis of why an application hung is more relevant to your problem. In kernel mode, **!analyze -hang** investigates locks that

the system holds and then scans the DPC queue chain. In user mode, !analyze -hang analyzes the thread stack to determine whether any threads are blocking other threads.

Before you run this extension in user mode, consider changing the current thread to the thread that you think has stopped responding (that is, hung). You should do this change because the exception might have altered the current thread to a different one.

Show parameter

-show `BugCheckCode` [`BugParameters`]

Displays information about the bug check specified by `BugCheckCode`. `BugParameters` specifies up to four bug check parameters separated by spaces. These parameters enable you to further refine your search.

Continue execution parameters

-c

Continues execution when the debugger encounters a known issue. If the issue isn't a known issue, the debugger remains broken into the target.

You can use the **-c** option with the following subparameters. These subparameters configure the list of known issues. They don't cause execution to occur by themselves. Until you run `!analyze -c **** -load` at least one time, `!analyze -c` has no effect.

-load `KnownIssuesFile`

Loads the specified known-issues file. `KnownIssuesFile` specifies the path and file name of this file. This file must be in XML format.

The list of known issues in the `KnownIssuesFile` file is used for all later **-c** commands until you use **-c -unload**, or until you use **-c -load** again (at which point the new data replaces the old data).

-unload

Unloads the current list of known issues.

-help

Displays help for the `!analyze -c` extension commands extension in the [Debugger command window](#).

XML load option parameters

-xml

Generates the analysis output in XML format.

-xmi

Adds module information to the xml output. This option requires -xml or -xmf.

-xcs

Adds the context and call stack frames to the xml output. This option requires -xml or -xmf.

-xmf `OutputXmlFile`

Writes the analysis to the specified `OutputXmlFile` in XML format. The file will be overwritten if it already exists. No analysis output will be generated to the console or log unless the -xml option is also specified.

DLL

`ext.dll`

Additional Information

For sample analysis of a user-mode exception and of a kernel-mode stop error (that is, crash), and for more information about how `!analyze` uses the triage.ini file, see [Using the !analyze extension](#).

Remarks

In user mode, `!analyze` displays information about the current exception.

In kernel mode, `!analyze` displays information about the most recent bug check. If a bug check occurs, the `!analyze` display is automatically generated. You can use `!analyze -v` to show additional information. If you want to see only the basic bug check parameters, you can use the [.bugcheck \(display bug check data\)](#) command.

For drivers that use User-Mode Driver Framework (UMDF) version 2.15 or later, `!analyze` provides information about UMDF verifier failures and unhandled exceptions. This functionality is available when performing live kernel-mode debugging and when

analyzing a user-mode memory dump file. For UMDF driver crashes, `!analyze` attempts to identify the responsible driver.

See also

- [Using the `!analyze` extension](#)
- [Bug check code reference](#)
- [Crash dump analysis using the Windows debuggers \(WinDbg\)](#)
- [Analyzing a kernel-mode dump file with WinDbg](#)

!asd

Article • 04/03/2024

The **!asd** extension displays a specified number of failure analysis entries from the data cache, starting at the specified address.

```
dbgcmd
  !asd Address DataUsed
```

Parameters

Address

Specifies the address of the first failure analysis entry to display.

DataUsed

Determines the number of tokens to display.

DLL

Ext.dll

Additional Information

You can use the [!dumpfa](#) extension to debug the [!analyze](#) extension.

Remarks

The **!asd** extension is useful only when you are debugging the [!analyze](#) extension.

Feedback

Was this page helpful?

 Yes

 No

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

!atom

Article • 10/25/2023

The **!atom** extension displays the formatted atom table for the specified atom or for all atoms of the current process.

```
dbgcmd
```

```
!atom [Address]
```

Parameters

Address

Specifies the hexadecimal virtual address of the atom to display. If you omit this parameter or specify zero, the atom table for the current process is displayed. This table lists all atoms for the process.

DLL

Exts.dll

Additional Information

For more information about atoms and atom tables, see the Microsoft Windows SDK documentation.

!bitcount

Article • 04/03/2024

The **!bitcount** extension counts the number of "1" bits in a memory range.

dbgcmd

```
!bitcount StartAddress TotalBits
```

Parameters

StartAddress

Specifies the starting address of the memory range whose "1" bits will be counted.

TotalBits

Specifies the size of the memory range, in bits.

-?

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Exts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!blackboxbsd

Article • 10/25/2023

The **!blackboxbsd** extension displays cached Boot Status Data (BSD) information when it is available in a kernel mode dump file. The information is retrieved from cached data in the kernel mode dump which was saved at the time the bugcheck occurred, and may not always be available.

Syntax

```
dbgcmd
!blackboxbsd
```

Parameters

None

DLL

ext.dll

Remarks

Driver developers can add secondary boot information to dump files. Driver developers (and the OS) can decide when to add this information to the dump file. This means that not all kernel mode dump files will contain secondary boot information. For more information, see [Writing a Bug Check Reason Callback Routine](#).

Example Command Output

```
dbgcmd
0: kd> !ext.blackboxbsd
Version: 136
Product type: 1

Auto advanced boot: FALSE
Advanced boot menu timeout: 30
Last boot succeeded: TRUE
Last boot shutdown: FALSE
```

Sleep in progress: FALSE

Power button timestamp: 0
System running: TRUE
Connected standby in progress: FALSE
User shutdown in progress: FALSE
System shutdown in progress: FALSE
Sleep in progress: 6
Connected standby scenario instance id: 5
Connected standby entry reason: 12
Connected standby exit reason: 31
System sleep transitions to on: 8
Last reference time: 0x1d3645f716b79e3
Last reference time checksum: 0xb6ae84b7
Last update boot id: 6

Boot attempt count: 1
Last boot checkpoint: TRUE
Checksum: 0x7f
Last boot id: 6
Last successful shutdown boot id: 2
Last reported abnormal shutdown boot id: 5

Error info boot id: 0
Error info repeat count: 0
Error info other error count: 0
Error info code: 0
Error info other error count: 0

Power button last press time: 0x1d365b105eb9e3b
Power button cumulative press count: 6
Power button last press boot id: 6
Power button last power watchdog stage: 0x20
Power button watchdog armed: FALSE
Power button shutdown in progress: FALSE
Power button last release time: 0x1d36112993b1191
Power button cumulative release count: 5
Power button last release boot id: 6
Power button error count: 0
Power button current connected standby phase: 1
Power button transition latest checkpoint id: 9
Power button transition latest checkpoint type: 0
Power button transition latest checkpoint sequence number: 77

!blackboxscm

Article • 10/25/2023

The **!blackboxscm** extension displays information from Service Control Manager (SCM) when it is available in a kernel mode dump file. The extension will display the names of any services which have outstanding Service Control Requests. The information is retrieved from cached data in the kernel mode dump which was saved at the time the bugcheck occurred, and may not always be available.

Syntax

```
dbgcmd
!blackboxscm
```

Parameters

None

DLL

ext.dll

Remarks

The commands have been dispatched to the LPHANDLER_FUNCTION_EX callback function of the specific service.

Outstanding requests such as SERVICE_CONTROL_SHUTDOWN or SERVICE_CONTROL_PRESHUTDOWN could be delaying the orderly shutdown or restart of the computer.

Example Command Output

In many dump files, just a single service is returned.

```
dbgcmd
2: kd> !ext.blackboxscm
      Name: gpsvc
```

Code: 15

The returned data provides information on two fields.

Name - The name of the service that was active at the time that the dump occurred.

Code - The Decimal value of the dwControl which is outstanding

In this example, the code 15 (or 0x0000000F) is defined as SERVICE_CONTROL_PRESHUTDOWN.

Multiple Services

When multiple services are listed, only the first service that is listed is typically of interest for failure analysis. This is because the SCM (Service Control Manager) waits serially for completion of these requests, so only the first service has actually received a control request.

For more information about SCM, see [Service Control Manager](#).

Additional Information

dwControl values are defined in winsvc.h and documented as parameters to [LPHANDLER_FUNCTION_EX callback function](#).

!chksym

Article • 10/25/2023

The **!chksym** extension tests the validity of a module against a symbol file.

dbgsyntax

!chksym Module [Symbol]

Parameters

Module

Specifies the name of the module by its name or base address.

Symbol

Specifies the name of a symbol file.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Vista and later	Dbghelp.dll

Remarks

If you do not specify a symbol file, the loaded symbol is tested. Otherwise, if you specify a .pdb or .dbg symbol file path, the loaded symbol is tested against the loaded module.

!chkimg

Article • 10/25/2023

The **!chkimg** extension detects corruption in the images of executable files by comparing them to the copy on a symbol store or other file repository.

dbgsyntax

```
!chkimg [Options] [-mmw LogFile LogOptions] [Module]
```

Parameters

Options Any combination of the following options:

-p ** *SearchPath***

Recursively searches *SearchPath* for the file before accessing the symbol server.

-f

Fixes errors in the image. Whenever the scan detects differences between the file on the symbol store and the image in memory, the contents of the file on the symbol store are copied over the image. If you are performing live debugging, you can create a dump file before you execute the **!chkimg -f** extension.

-nar

Prevents the mapped image of the file on the symbol server from being moved. By default, when the copy of the file is located on the symbol server and mapped into memory, **!chkimg** moves the image of the file on the symbol server. However, if you use the **-nar** option, the image of the file from the server is not moved.

The executable image that is already in memory (that is, the one that is being scanned) is moved, because the debugger always relocates images that it loads.

This switch is useful only if the operating system already moved the original image. If the image has not been moved, **!chkimg** and the debugger will move the image. Use of this switch is rare.

-ss ** *SectionName***

Limits the scan to those sections whose names contain the string *SectionName*. The scan will include any non-discardable section whose name contains this string. *SectionName* is case sensitive and cannot exceed 8 characters.

-as

Causes the scan to include all sections of the image except discardable sections. By default, (if you do not use **-as** or **-ss**), the scan skips sections that are writeable, sections that are not executable, sections that have "PAGE" in their name, and discardable sections.

-r ** StartAddress **** EndAddress**

Limits the scan to the memory range that begins with *StartAddress* and ends with *EndAddress*. Within this range, any sections that would typically be scanned are scanned. If a section partially overlaps with this range, only that part of the section that overlaps with this range is scanned. The scan is limited to this range even if you also use the **-as** or **-ss** switch.

-nospec

Causes the scan to include the reserved sections of Hal.dll and Ntoskrnl.exe. By default, **!chkimg** does not check certain parts of these files.

-noplock

Displays areas that mismatch by having a byte value of 0x90 (a **nop** instruction) and a byte value of 0xF0 (a **lock** instruction). By default, these mismatches are not displayed.

-np

Causes patched instructions to be recognized.

-d

Displays a summary of all mismatched areas while the scan is occurring. For more information about this summary text, see the Remarks section.

-db

Displays mismatched areas in a format that is similar to the [db debugger command](#). Therefore, each display line shows the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. All nonprintable characters, such as carriage returns and line feeds, are displayed as periods (.). The mismatched bytes are marked by an asterisk (*).

-lo ** lines**

Limits the number of output lines that **-d** or **-db** display to the *lines* number of lines.

-v

Displays verbose information.

-mmw

Creates a log file and records the activity of **!chkimg** in this file. Each line of the log file

represents a single mismatch.

LogFile

Specifies the full path of the log file. If you specify a relative path, the path is relative to the current path.

LogOptions

Specifies the contents of the log file. *LogOptions* is a string that consists of a concatenation of various letters. Each line in the log file contains several columns that are separated by commas. These columns include the items that the following option letters specify, in the order that the letters appear in the *LogOptions* string. You can include the following options multiple times. You must include at least one option.

Log option	Information included in the log file
v	The virtual address of the mismatch
r	The offset (relative address) of the mismatch within the module
s	The symbol that corresponds to the address of the mismatch
S	The name of the section that contains the mismatch
e	The correct value that was expected at the mismatch location
w	The incorrect value that was at the mismatch location

LogOptions can also include some, or none, of the following additional options.

Log option	Effect
o	If a file that has the name <i>LogFile</i> already exists, the existing file is overwritten. By default, the debugger appends new information to the end of any existing file.
t <i>String</i>	Adds an extra column to the log file. Each entry in this column contains <i>String</i> . The <i>tString</i> option is useful if you are appending new information to an existing log file and you have to distinguish the new records from the old. You cannot add space between t and <i>String</i> . If you use the <i>tString</i> option, it must be the final

Log option	Effect
	option in <i>LogOptions</i> , because <i>String</i> is taken to include all of the characters that are present before the next space.

For example, if *LogOptions* is **rSewo**, each line of the log file contains the relative address and section name of the mismatch location and the expected and actual values at that location. This option also causes any previous file to be overwritten. You can use the **-mmw** switch multiple times if you want to create several log files that have different options. You can create up to 10 log files at the same time.

Module

Specifies the module to check. *Module* can be the name of the module, the starting address of the module, or any address that is contained in the module. If you omit *Module*, the debugger uses the module that contains the current instruction pointer.

DLL

Windows XP and later	Ext.dll
----------------------	---------

Remarks

When you use **!chkimg**, it compares the image of an executable file in memory to the copy of the file that resides on a symbol store.

All sections of the file are compared, except for sections that are discardable, that are writeable, that are not executable, that have "PAGE" in their name, or that are from INITKDBG. You can change this behavior can by using the **-ss**, **-as**, or **-r** switches.

!chkimg displays any mismatch between the image and the file as an image error, with the following exceptions:

- Addresses that are occupied by the Import Address Table (IAT) are not checked.
- Certain specific addresses in Hal.dll and Ntoskrnl.exe are not checked, because certain changes occur when these sections are loaded. To check these addresses, include the **-nospec** option.
- If the byte value 0x90 is present in the file, and if the value 0xF0 is present in the corresponding byte of the image (or vice versa), this situation is considered a match. Typically, the symbol server holds one version of a binary that exists in both uniprocessor and multiprocessor versions. On an x86-based processor, the **lock**

instruction is 0xF0, and this instruction corresponds to a **nop** (0x90) instruction in the uniprocessor version. If you want **!chkimg** to display this pair as a mismatch, set the **-noplock** option.

Note If you use the **-f** option to fix image mismatches, **!chkimg** fixes only those mismatches that it considers to be errors. For example, **!chkimg** does not change an 0x90 byte to an 0xF0 byte unless you include **-noplock**.

When you include the **-d** option, **!chkimg** displays a summary of all mismatched areas while the scan is occurring. Each mismatch is displayed on two lines. The first line includes the start of the range, the end of the range, the size of the range, the symbol name and offset that corresponds to the start of the range, and the number of bytes since the last error (in parentheses). The second line is enclosed in brackets and includes the hexadecimal byte values that were expected, a colon, and then the hexadecimal byte values that were actually encountered in the image. If the range is longer than 8 bytes, only the first 8 bytes are shown before the colon and after the colon. The following example shows this situation.

```
dbgcmd  
be000015-be000016 2 bytes - win32k!VeryUsefulFunction+15 (0x8)  
[ 85 dd:95 23 ]
```

Occasionally, a driver alters part of the Microsoft Windows kernel by using hooks, redirection, or other methods. Even a driver that is no longer on the stack might have altered part of the kernel. You can use the **!chkimg** extension as a file comparison tool to determine which parts of the Windows kernel (or any other image) are being altered by drivers and exactly how the parts are being changed. This comparison is most effective on full dump files.

Check each loaded module

You can also use **!chkimg** together with the **[!for_each_module](#)** extension to check the image of each loaded module. The following example shows this situation.

```
dbgcmd  
!for_each_module !chkimg @#ModuleName
```

!analyze example

Suppose that you encounter a bug check, for example, and begin by using **[!analyze](#)**.

```
dbgcmd
```

```
kd> !analyze
.....
BugCheck 1000008E, {c0000005, bf920e48, baf75b38, 0}
Probably caused by : memory_corruption
CHKIMG_EXTENSION: !chkimg !win32k
....
```

In this example, the **!analyze** output suggests that memory corruption has occurred and includes a **CHKIMG_EXTENSION** line that suggests that Win32k.sys could be the corrupted module. (Even if this line is not present, you might consider possible corruption in the module on top of the stack.) Start by using **!chkimg** without any switches, as the following example shows.

```
dbgcmd
```

```
kd> !chkimg win32k
Number of different bytes for win32k: 31
```

The following example shows that there are indeed memory corruptions. Use **!chkimg -d** to display all of the errors for the Win32k module.

```
dbgcmd
```

```
kd> !chkimg win32k -d
bf920e40-bf920e46 7 bytes - win32k!HFDBASIS32::vSteadyState+1f
[ 78 08 d3 78 0c c2 04:00 00 00 00 00 01 00 ]
bf920e48-bf920e5f 24 bytes - win32k!HFDBASIS32::vHalveStepSize (+0x08)
[ 8b 51 0c 8b 41 08 56 8b:00 00 00 00 00 00 00 00 ]
Number of different bytes for win32k: 31
```

When you try to disassemble the corrupted image of the second section that is listed, the following output might occur.

```
dbgcmd
```

```
kd> u win32k!HFDBASIS32::vHalveStepSize
win32k!HFDBASIS32::vHalveStepSize:
bf920e48 0000          add     [eax],al
bf920e4a 0000          add     [eax],al
bf920e4c 0000          add     [eax],al
bf920e4e 0000          add     [eax],al
bf920e50 7808          js      win32k!HFDBASIS32::vHalveStepSize+0x12
(bf920e5a)
bf920e52 d3780c        sar     dword ptr [eax+0xc],cl
```

```
bf920e55 c20400      ret     0x4
bf920e58 8b510c      mov     edx,[ecx+0xc]
```

Then, use `!chkimg -f` to fix the memory corruption.

```
dbgcmd

kd> !chkimg win32k -f
Warning: Any detected errors will be fixed to what we expect!
Number of different bytes for win32k: 31 (fixed)
```

Now you can disassemble the corrected view and see the changes that you have made.

```
dbgcmd

kd> u win32k!HFDBASIS32::vHalveStepSize
win32k!HFDBASIS32::vHalveStepSize:
bf920e48 8b510c      mov     edx,[ecx+0xc]
bf920e4b 8b4108      mov     eax,[ecx+0x8]
bf920e4e 56           push    esi
bf920e4f 8b7104      mov     esi,[ecx+0x4]
bf920e52 03c2           add    eax,edx
bf920e54 c1f803      sar     eax,0x3
bf920e57 2bf0           sub    esi,eax
bf920e59 d1fe           sar    esi,1
```

Investigate storage and memory corruption

Random file and memory corruption can be difficult to investigate. One tool to consider in some situations is enabling additional memory checking, for example using driver verifier. For information about Driver Verifier, see [Driver Verifier](#).

For testing physical memory use the Windows Memory Diagnostics tool. Its use and other general techniques are described in [Blue Screen Data](#).

Use the scan disk utility to identify file system errors. Select and hold (or right-click) on the drive you want to scan and select **Properties**. Select **Tools**. Select the **Check now** button.

!cppexpr

Article • 04/03/2024

The **!cppexpr** extension displays the contents of a C++ exception record.

dbgsyntax

!cppexpr Address

Parameters

Address

Specifies the address of the C++ exception record to display.

DLL

Ext.dll

Additional Information

For more information about exceptions, see [Controlling Exceptions and Events](#), the Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Use the [.exr](#) command to display other exception records.

Remarks

The **!cppexpr** extension displays information that is related to a C++ exception that the target encounters, including the exception code, the address of the exception, and the exception flags. This exception must be one of the standard C++ exceptions that are defined in Msvcrt.dll.

You can typically obtain the *Address* parameter by using the [!analyze -v](#) command.

The **!cppexpr** extension is useful for determining the type of a C++ exception.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!cpuid

Article • 04/03/2024

The **!cpuid** extension displays information about the processors on the system.

dbgsyntax

```
!cpuid [Processor]
```

Parameters

Processor

Specifies the processor whose information will be displayed. If you omit this parameter, all processors are displayed.

DLL

Ext.dll

Additional Information

For more information about how to debug multiprocessor computers, see

[Multiprocessor Syntax](#).

Remarks

The **!cpuid** extension works during live user-mode or kernel-mode debugging, local kernel debugging, and debugging of dump files. However, user-mode minidump files contain only information about the active processor.

If you are debugging in user mode, the **!cpuid** extension describes the computer that the target application is running on. In kernel mode, it describes the target computer.

The following example shows this extension.

dbgcmd

```
kd> !cpuid
CP F/M/S Manufacturer MHz
```

0	6,5,1	GenuineIntel	700
1	8,1,5	AuthenticAMD	700

The **CP** column gives the processor number. (These numbers are always sequential, starting with zero). The **Manufacturer** column specifies the processor manufacturer. The **MHz** column specifies the processor speed, if it is available.

For an x86-based processor or an x64-based processor, the **F** column displays the processor family number, the **M** column displays the processor model number, and the **S** column displays the stepping size.

Feedback

Was this page helpful?

 Yes

 No

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

!cs

Article • 10/25/2023

The **!cs** extension displays one or more critical sections or the whole critical section tree.

dbgsyntax

```
!cs [-s] [-l] [-o]
!cs [-s] [-o] Address
!cs [-s] [-l] [-o] StartAddress EndAddress
!cs [-s] [-o] -d InfoAddress
!cs [-s] -t [TreeAddress]
!cs -?
```

Parameters

Parameter	Description
-s	Displays each critical section's initialization stack trace, if this information is available.
-l	Display only the locked critical sections.
-o	Displays the owner's stack for any locked critical section that is being displayed.
Address	Specifies the address of the critical section to display. If you omit this parameter, the debugger displays all critical sections in the current process.
StartAddress	Specifies the beginning of the address range to search for critical sections.
EndAddress	Specifies the end of the address range to search for critical sections.
-d	Displays critical sections that are associated with DebugInfo.
InfoAddress	Specifies the address of the DebugInfo.
-t	Displays a critical section tree. Before you can use the -t option, you must activate Application Verifier for the target process and select the Check lock usage option.
TreeAddress	Specifies the address of the root of the critical section tree. If you omit this parameter or specify zero, the debugger displays the critical section tree for the current process.
-?	Displays some Help text for this extension in the Debugger Command window .

DLL

Additional Information

For other commands and extensions that can display critical section information, see [Displaying a Critical Section](#). For more information about critical sections, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!cs** extension requires full symbols (including type information) for the process that is being debugged and for Ntdll.dll.

The following examples shows you how to use **!cs**. The following command displays information about the critical section at address 0x7803B0F8 and shows its initialization stack trace.

```
dbgcmd

0:001> !cs -s 0x7803B0F8
CriticalSection      = 0x7803B0F8 (MSVCRT!__app_type+0x4)
DebugInfo           = 0x6A262080
NOT LOCKED
LockSemaphore      = 0x0
SpinCount           = 0x0

Stack trace for DebugInfo = 0x6A262080:

0x6A2137BD: ntdll!RtlInitializeCriticalSectionAndSpinCount+0x9B
0x6A207A4C: ntdll!LdrpCallInitRoutine+0x14
0x6A205569: ntdll!LdrpRunInitializeRoutines+0x1D9
0x6A20DCE1: ntdll!LdrpInitializeProcess+0xAE5
```

The following command displays information about the critical section whose DebugInfo is at address 0x6A262080.

```
dbgcmd

0:001> !cs -d 0x6A262080
DebugInfo           = 0x6A262080
CriticalSection      = 0x7803B0F8 (MSVCRT!__app_type+0x4)
NOT LOCKED
LockSemaphore      = 0x0
SpinCount           = 0x0
```

The following command displays information about all of the active critical sections in the current process.

```
dbgcmd

## 0:001> !cs

DebugInfo      = 0x6A261D60
CriticalSection = 0x6A262820 (ntdll!RtlCriticalSectionLock+0x0)
LOCKED
LockCount      = 0x0
OwningThread   = 0x460
RecursionCount = 0x1
LockSemaphore  = 0x0
## SpinCount    = 0x0

DebugInfo      = 0x6A261D80
CriticalSection = 0x6A262580 (ntdll!DeferedCriticalSection+0x0)
NOT LOCKED
LockSemaphore  = 0x7FC
## SpinCount    = 0x0

DebugInfo      = 0x6A262600
CriticalSection = 0x6A26074C (ntdll!LoaderLock+0x0)
NOT LOCKED
LockSemaphore  = 0x0
## SpinCount    = 0x0

DebugInfo      = 0x77fbde20
CriticalSection = 0x77c8ba60 (GDI32!semColorSpaceCache+0x0)
LOCKED
LockCount      = 0x0
OwningThread   = 0x000000dd8
RecursionCount = 0x1
LockSemaphore  = 0x0
## SpinCount    = 0x000000000

...
```

The following command displays the critical section tree.

```
dbgcmd

0:001> !cs -t

Tree root 00bb08c0

Level      Node      CS      Debug  InitThr EnterThr  WaitThr TryEnThr
LeaveThr  EnterCnt  WaitCnt
##
```

	0	00bb08c0	77c7e020	77fbcae0	4c8	4c8	0	0
4c8	c	0						
1	00dd6fd0	0148cf8	01683fe0		4c8	4c8	0	0
2	0							4c8
2	00bb0aa0	008e8b84	77fbcc20		4c8	0	0	0
0	0							
3	00bb09e0	008e8704	77fbcba0		4c8	0	0	0
0	0	0						
4	00bb0a40	008e8944	77fbcbe0		4c8	0	0	0
0	0	0						
5	00bb0a10	008e8824	77fbcbc0		4c8	0	0	0
0	0	0						
5	00bb0a70	008e8a64	77fbcc00		4c8	0	0	0
0	0	0						
3	00bb0b00	008e8dc4	77fbcc60		4c8	0	0	0
0	0	0						
4	00bb0ad0	008e8ca4	77fbcc40		4c8	0	0	0
0	0	0						
4	00bb0b30	008e8ee4	77fbcc80		4c8	0	0	0
0	0	0						
5	00dd4fd0	0148afe4	0167ffe0		4c8	0	0	0
0	0	0						
2	00bb0e90	77c2da98	00908fe0		4c8	4c8	0	0
4c8	3a	0						
3	00bb0d70	77c2da08	008fcfe0		4c8	0	0	0
0	0							

The following items appear in this !cs -t display:

- **InitThr** is the thread ID for the thread that initialized the CS.
- **EnterThr** is the ID of the thread that called **EnterCriticalSection** last time.
- **WaitThr** is the ID of the thread that found the critical section that another thread owned and waited for it last time.
- **TryEnThr** is the ID of the thread that called **TryEnterCriticalSection** last time.
- **LeaveThr** is the ID of the thread that called **LeaveCriticalSection** last time
- **EnterCnt** is the count of **EnterCriticalSection**.
- **WaitCnt** is the contention count.

See also

[!ntsdexts.locks](#)

[Displaying a Critical Section](#)

Critical Section Time Outs (user mode)

!cxr

Article • 11/02/2023

The **!cxr** extension command is obsolete. Use the [.cxr \(Display Context Record\)](#) command instead.

!dh

Article • 10/25/2023

The **!dh** extension displays the headers for the specified image.

dbgcmd

```
!dh [Options] Address  
!dh -h
```

Parameters

Options Any one of the following options:

-f

Displays file headers.

-s

Displays section headers.

-a

Displays all header information.

Address

Specifies the hexadecimal address of the image.

-h

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Windows 2000	Dbghelp.dll Kdextx86.dll Ntsdexts.dll
Windows XP and later	Dbghelp.dll

Remarks

The **!lmi** extension extracts the most important information from the image header and displays it in a concise summary format. That extension is frequently more useful than **!dh**.

!dlls

Article • 04/03/2024

The **!dlls** extension displays the table entries of all loaded modules or all modules that a specified thread or process are using.

dbgcmd

```
!dlls [Options] [LoaderEntryAddress]  
!dlls -h
```

Parameters

Options Specifies the level of output. This parameter can be any combination of the following values:

-f

Displays file headers.

-s

Displays section headers.

-a

Displays complete module information. (This option is equivalent to **-f -s**.)

-c ** ModuleAddress**

Displays the module that contains *ModuleAddress*.

-i

Sorts the display by initialization order.

-l

Sorts the display by load order. This situation is the default.

-m

Sorts the display by memory order.

-v

Displays version information. This information is taken from the resource section of each module.

LoaderEntryAddress

Specifies the address of the loader entry for a module. If you include this parameter, the

debugger displays only this specific module.

-h

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Exts.dll

Remarks

The module listing includes all entry points into each module.

The .dlls extension works only in live debugging (not with crash dump analysis).

In kernel mode, this extension displays the modules for the current [process context](#). You cannot use !dlls together with a system process or the idle process.

The following examples shows you how to use the !dlls extension.

```
dbgcmd

kd> !dlls -c 77f60000
Dump dll containing 0x77f60000:

0x00091f38: E:\WINDOWS\System32\ntdll.dll
    Base 0x77f60000 EntryPoint 0x00000000 Size      0x00097000
    Flags 0x00004004 LoadCount   0x0000ffff TlsIndex 0x00000000
          LDRP_IMAGE_DLL
          LDRP_ENTRY_PROCESSED

kd> !dlls -a 91ec0

0x00091ec0: E:\WINDOWS\system32\winmine.exe
    Base 0x01000000 EntryPoint 0x01003e2e Size      0x00020000
    Flags 0x00005000 LoadCount   0x0000ffff TlsIndex 0x00000000
          LDRP_LOAD_IN_PROGRESS
          LDRP_ENTRY_PROCESSED

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
  14C machine (i386)
  3 number of sections
3A98E856 time date stamp Sun Feb 25 03:11:18 2001

  0 file pointer to symbol table
  0 number of symbols
E0 size of optional header
10F characteristics
```

```
Relocations stripped
Executable
Line numbers stripped
Symbols stripped
32 bit word machine
```

OPTIONAL HEADER VALUES

```
10B magic #
7.00 linker version
3A00 size of code
19E00 size of initialized data
    0 size of uninitialized data
3E2E address of entry point
1000 base of code
    ----- new -----
01000000 image base
1000 section alignment
200 file alignment
    2 subsystem (Windows GUI)
5.01 operating system version
5.01 image version
4.00 subsystem version
20000 size of image
    400 size of headers
21970 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
01000100 Opt Hdr
    0 [      0] address [size] of Export Directory
    40B4 [      B4] address [size] of Import Directory
    6000 [ 19170] address [size] of Resource Directory
    0 [      0] address [size] of Exception Directory
    0 [      0] address [size] of Security Directory
    0 [      0] address [size] of Base Relocation Directory
    11B0 [      1C] address [size] of Debug Directory
    0 [      0] address [size] of Description Directory
    0 [      0] address [size] of Special Directory
    0 [      0] address [size] of Thread Storage Directory
    0 [      0] address [size] of Load Configuration Directory
    258 [      A8] address [size] of Bound Import Directory
    1000 [    1B0] address [size] of Import Address Table Directory
    0 [      0] address [size] of Reserved Directory
    0 [      0] address [size] of Reserved Directory
    0 [      0] address [size] of Reserved Directory
```

SECTION HEADER #1

```
.text name
3992 virtual size
1000 virtual address
3A00 size of raw data
400 file pointer to raw data
    0 file pointer to relocation table
```

```

    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
60000020 flags
    Code
    (no align specified)
    Execute Read

Debug Directories(1)
  Type      Size     Address   Pointer
  cv          1c        13d0       7d0      Format: NB10, 3a98e856, 1,
winmi
ne.pdb

SECTION HEADER #2
  .data name
    BB8 virtual size
    5000 virtual address
    200 size of raw data
    3E00 file pointer to raw data
      0 file pointer to relocation table
      0 file pointer to line numbers
      0 number of relocations
      0 number of line numbers
C0000040 flags
    Initialized Data
    (no align specified)
    Read Write

SECTION HEADER #3
  .rsrc name
  19170 virtual size
  6000 virtual address
  19200 size of raw data
  4000 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
40000040 flags
    Initialized Data
    (no align specified)
    Read Only

```

Feedback

Was this page helpful?

Yes

No

!dml_proc

Article • 10/25/2023

The **!dml_proc** extension displays a list of processes and provides links for obtaining more detailed information about processes.

```
dbgcmd
!dml_proc
```

Remarks

The following image shows a portion of the output displayed by **!dml_proc**.

Address	Process Name
fffffa80`0639b600	svchost.exe
fffffa80`06387080	conhost.exe
fffffa80`06301b00	wscript.exe
fffffa80`064c05c0	SearchIndexer.exe
fffffa80`05fbc740	svchost.exe
fffffa80`05f0f5c0	wmpnetwk.exe
fffffa80`06493080	csrss.exe
fffffa80`05e57b00	winlogon.exe
fffffa80`06889080	dwm.exe
fffffa80`045805c0	taskhost.exe
fffffa80`0696cb00	explorer.exe
fffffa80`04e2b700	mobsync.exe
fffffa80`04589700	wlrmldr.exe
fffffa80`045c9080	LogonUI.exe

In the preceding output, the process addresses are links that you can click to see more detailed information. For example, if you click **fffffa80`04e2b700** (the address for **mobsync.exe**), you will see detailed information about the **mobsync.exe** process as shown in the following image.

The screenshot shows the WinDbg command window with the following output:

```
!dml_proc 0xffff... - Kernel 'net:port=50000,key=*****' - WinDbg:6.13.0006.10
Command: !dml_proc 0xfffffa8004e2b700
Address          PID  Image file name
fffffa80`04e2b700 1e38  mobsync.exe      Full details
Select user-mode state  Release user-mode state
Browse kernel module list  Browse user module list
Browse full module list

Threads:
Address          TID
fffffa80`04e05240 1e3c
fffffa80`0640e5c0 1e54
fffffa80`065fcb40 1e58
fffffa80`03a37b40 1ab4
fffffa80`06fae880 17c8
fffffa80`038edb40 1668
```

The preceding output, which describes an individual process, contains links that you can click to explore the process and its threads in more detail.

See also

[Debugger Markup Language Commands](#)

!dumpfa

Article • 04/03/2024

The **!dumpfa** extension displays the contents of a failure analysis entry.

```
dbgcmd
```

```
!dumpfa Address
```

Parameters

Address

Specifies the address of the failure analysis entry that is displayed.

DLL

Ext.dll

Remarks

The **.dumpfa** extension is useful only to debug the [!analyze](#) extension, as the following example shows.

```
dbgcmd
```

```
0:000> !dumpfa 0x00a34140
DataUsed 3b0
Type = DEBUG_FLR_MARKER_BUCKET 00010016 - Size = 9
Type = DEBUG_FLR_MARKER_FILE 0001000d - Size = 16
Type = DEBUG_FLR_SYSXML_LOCALEID 00004200 - Size = 4
Type = DEBUG_FLR_SYSXML_CHECKSUM 00004201 - Size = 4
Type = DEBUG_FLR_READ_ADDRESS 0000000e - Size = 8
Type = DEBUG_FLR_FAULTING_IP 80000000 - Size = 8
Type = DEBUG_FLR_MM_INTERNAL_CODE 00001004 - Size = 8
Type = DEBUG_FLR_CPU_MICROCODE_VERSION 0000301f - Size = 28
Type = DEBUG_FLR_CUSTOMER_CRASH_COUNT 0000300b - Size = 8
Type = DEBUG_FLR_DEFAULT_BUCKET_ID 00010008 - Size = 12
Type = DEBUG_FLR_BUGCHECK_STR 00000600 - Size = 5
Type = DEBUG_FLR_LAST_CONTROL_TRANSFER 0000000a - Size = 18
Type = DEBUG_FLR_TRAP_FRAME c0000002 - Size = 8
Type = DEBUG_FLR_STACK_TEXT 00010005 - Size = 1fb
Type = DEBUG_FLR_STACK_COMMAND 00010004 - Size = 17
Type = DEBUG_FLR_OS_BUILD_NAME 0000301e - Size = 9
Type = DEBUG_FLR_MODULE_NAME 00010006 - Size = 8
```

```
Type = DEBUG_FLR_IMAGE_NAME 00010001 - Size = c
Type = DEBUG_FLR_IMAGE_TIMESTAMP 80000002 - Size = 8
```

You can also use the [!asd](#) extension to debug the [!analyze](#) extension.

Feedback

Was this page helpful?

 Yes

 No

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

!envvar

Article • 04/03/2024

The **!envvar** extension displays the value of the specified environment variable.

```
dbgcmd
```

```
!envvar Variable
```

Parameters

Variable

Specifies the environment variable whose value is displayed. *Variable* is not case sensitive.

DLL

Exts.dll

Additional Information

For more information about environment variables, see [Environment Variables](#) and the Microsoft Windows SDK documentation.

Remarks

The **!envvar** extension works both in user mode and in kernel mode. However, in kernel mode, when you set the idle thread as the current process, the pointer to the Process Environment Block (PEB) is **NULL**, so it fails. In kernel mode, the **!envvar** extension displays the environment variables on the target computer, as the following example shows.

```
dbgcmd
```

```
0:000> !envvar _nt_symbol_path
      _nt_symbol_path =
      srv*C:\mysyms*https://msdl.microsoft.com/download/symbols
```

Feedback

Was this page helpful?

 Yes

 No

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

!error

Article • 04/03/2024

The **!error** extension decodes and displays information about an error value.

```
dbgcmd
```

```
!error Value [Flags]
```

Parameters

Value

Specifies one of the following error codes:

- Win32
- Winsock
- NTSTATUS
- NetAPI

Flags

If *Flags* is set to 1, the error code is read as an NTSTATUS code.

DLL

Ext.dll

Remarks

The following example shows you how to use **!error**.

```
dbgcmd
```

```
0:000> !error 2
Error code: (Win32) 0x2 (2) - The system cannot find the file specified.
0:000> !error 2 1
Error code: (NTSTATUS) 0x2 - STATUS_WAIT_2
```

Feedback

Was this page helpful?

 Yes

 No

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

!exchain

Article • 04/03/2024

The **!exchain** extension displays the current exception handler chain.

dbgcmd

```
!exchain [Options]
```

Parameters

Options One of the following values:

/c

Displays information that is relevant for debugging a C++ **try/catch** exception, if such an exception is detected.

/C

Displays information that is relevant for debugging a C++ **try/catch** exception, even when such an exception has not been detected.

/f

Displays information that is obtained by walking the CRT function tables, even if a CRT exception handler was not detected.

DLL

Ext.dll

The **!exchain** extension is available only for an x86-based target computer.

Remarks

The **!exchain** extension displays the list of exception handlers for the current thread.

The list begins with the first handler on the chain (the one that is given the first opportunity to handle an exception) and continues on to the end. The following example shows this extension.

dbgcmd

```
0:000> !exchain
0012fea8: Prymes!_except_handler3+0 (00407604)
    CRT scope 0, filter: Prymes!dzExcepError+e6 (00401576)
        func:  Prymes!dzExcepError+ec (0040157c)
0012ffb0: Prymes!_except_handler3+0 (00407604)
    CRT scope 0, filter: Prymes!mainCRTStartup+f8 (004021b8)
        func:  Prymes!mainCRTStartup+113 (004021d3)
0012ffe0: KERNEL32!GetThreadContext+1c (77ea1856)
```

Feedback

Was this page helpful?

 Yes

 No

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

!exr

Article • 10/25/2023

The **!exr** extension command is obsolete. Use the [.exr \(Display Exception Record\)](#) command instead.

!findxmldata

Article • 04/03/2024

The **!findxmldata** extension retrieves XML data from a CAB file that contains a kernel-mode Small Memory Dump file.

dbgcmd

```
!findxmldata [ -d DeviceName | -h HwId ]
!findxmldata -r Driver
!findxmldata -chksum [ -z CabFile ]
!findxmldata -v
```

Parameters

-d *DeviceName*

Displays all devices whose device name contains the string that *DeviceName* specifies.

-h *Hwid*

Displays all devices whose hardware IDs contain the string that *Hwid* specifies. If you use both **-d** and **-h**, the debugger displays only those devices that satisfy both matches.

-r *Driver*

Displays information about the driver that the *Driver* parameter specifies, including all devices that use this driver.

-chksum

Displays the XML file's checksum.

-z *CabFile*

Enables you to perform a checksum on the CAB file that the *CabFile* parameter specifies, instead of on the default Sysdata.xml file.

-v

Displays system version information.

DLL

Ext.dll

Additional Information

The **!findxmldata** extension works only on a kernel-mode Small Memory Dump file that is stored in a CAB file.

For more information about how to put dump files into CAB files, see [.dumpcab \(Create Dump File CAB\)](#). For information more about how to debug a kernel-mode dump file, including dump files that are stored inside CAB files, see [Analyzing a Kernel-Mode Dump File](#).

Remarks

The **!findxmldata** extension retrieves data from the Sysdata.xml file that is stored in a CAB file that contains a kernel-mode [Small Memory Dump](#) file.

When you do not use any options, the extension displays all devices.

The following examples show you how to use **!findxmldata**.

```
dbgcmd

kd> !findxmldata -v
SYSTEM Info:
OSVER: 5.1.2600 2.0
OSLANGUAGE: 2052
OSNAME: Microsoft Windows XP Home Edition
kd> !findxmldata -d MIDI
Node DEVICE
DESCRIPTION      : MPU-401 Compatible MIDI Device
HARDWAREID      : ACPI\PNPB006
SERVICE          : ms_mpu401
DRIVER           : msmpu401.sys

kd> !findxmldata -r msmpu
Node DRIVER
FILENAME        : msmpu401.sys
FILESIZE         : 2944
CREATIONDATE    : 05-06-2005 09:18:34
VERSION          : 5.1.2600.0
MANUFACTURER    : Microsoft Corporation
PRODUCTNAME     : Microsoft® Windows® Operating System
Node DEVICE
DESCRIPTION      : MPU-401 Compatible MIDI Device
HARDWAREID      : ACPI\PNPB006
SERVICE          : ms_mpu401
DRIVER           : msmpu401.sys

kd> !findxmldata -h PCI\VEN_8086&DEV_24C3&SUBSYS_24C28086
Node DEVICE
DESCRIPTION      : Intel(R) 82801DB/DBM SMBus Controller - 24C3
HARDWAREID      : PCI\VEN_8086&DEV_24C3&SUBSYS_24C28086&REV_01
kd> !findxmldata -h USB\ROOT_HUB&VID8086&PID24C4&REV0001
```

```
Node DEVICE
    DESCRIPTION      : USB Root Hub
HARDWAREID      : USB\ROOT_HUB&VID8086&PID24C4&REV0001
    SERVICE        : usbhub
    DRIVER         : usbhub.sys

kd> !findxmldata -h ACPI\PNPB006
Node DEVICE
    DESCRIPTION      : MPU-401 Compatible MIDI Device
HARDWAREID      : ACPI\PNPB006
    SERVICE        : ms_mpu401
    DRIVER         : msmpu401.sys
```

Feedback

Was this page helpful?

 Yes

 No

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

!for_each_frame

Article • 04/03/2024

The **!for_each_frame** extension executes a debugger command one time for each frame in the stack of the current thread.

dbgcmd

```
!for_each_frame ["CommandString"]
!for_each_frame -?
```

Parameters

CommandString

Specifies the debugger commands to execute one time for each frame. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands within *CommandString* cannot contain quotation marks. If you want to refer to the index of the current frame within *CommandString*, use the @\$frame pseudoregister.

-?

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Ext.dll

Additional Information

For more information about the local context, see [Changing Contexts](#).

Remarks

If you do not specify any arguments, the **!for_each_frame** extension displays a list of all frames and their frame indexes. For a more detailed list of all frames, use the [k \(Display Stack Backtrace\)](#) command.

The **k** command walks up to 256 frames. For each enumerated frame, that frame temporarily becomes the current local context (similar to the [.frame \(Set Local Context\)](#) command). After the context has been set, *CommandString* is executed. After all frames have been used, the local context is reset to the value that it had before you used the **!for_each_frame** extension.

If you include *CommandString*, the debugger displays the frame and its index before the command is executed for that frame.

The following command displays all local variables for the current stack.

```
dbgcmd  
!for_each_frame !for_each_local dt @#Local
```

Feedback

Was this page helpful?

 Yes

 No

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

!for_each_function

Article • 10/25/2023

The **!for_each_function** extension executes a debugger command for each function, in a specified module, whose name matches a specified pattern.

dbgcmd

```
!for_each_function -m:ModuleName -p:Pattern -c:CommandString  
!for_each_function -?
```

Parameters

-m:ModuleName

Specifies the module name. This name is typically the file name without the file name extension. In some cases, the module name differs significantly from the file name.

-p:Pattern

Specifies the pattern to be matched.

-c:CommandString

Specifies the debugger command to execute for each function, in the specified module, that matches the pattern.

You can use the following aliases in *CommandString*.

Alias	Data type	Value
@#SymbolName	string	The symbol name.
@#SymbolAddress	ULONG64	The symbol address.
@#ModName	string	The module name.
@#FunctionName	string	The function name.

-?

Displays help for this extension.

DLL

Ext.dll

Remarks

The following example shows how to list all function names, in the PCI module, that match the pattern *read*.

```
dbgcmd

1: kd> !for_each_function -m:pci -p:*read* -c:.echo @#FunctionName

PciReadDeviceConfig
PciReadDeviceSpace
PciReadSlotIdData
PciExternalReadDeviceConfig
PiRegStateReadStackCreationSettingsFromKey
VmProxyReadDevicePathsFromRegistry
PpRegStateReadCreateClassCreationSettings
ExpressRootPortReadConfigSpace
PciReadRomImage
PciDevice_ReadConfig
PciReadDeviceConfigEx
PciReadSlotConfig
```

If an alias is not preceded and followed by a blank space, you must put the alias inside the [\\${} Alias Interpreter](#) token.

The following example shows how to list all symbols, in all modules, whose function names match the pattern *CreateFile*. The alias @#ModuleName is not preceded by a blank space. Therefore, it is put inside the [\\${} Alias Interpreter](#) token.

Note Do not confuse the @#ModuleName alias with the @#ModName alias. The @#ModuleName alias belongs to the [!for_each_module](#) extension, and the @#ModName alias belongs to the [!for_each_function](#) extension.

```
dbgcmd

1: kd> !for_each_module !for_each_function -m:${@#ModuleName} -
p:*CreateFile* -c:.echo @#SymbolName
nt!BiCreateFileDeviceElement
nt!NtCreateFile
...
Wdf01000!FxFileObject::_CreateFileObject
fltmgr!FltCreateFileEx2$fin$0
fltmgr!FltCreateFileEx2
...
Ntfs!TxfIoCreateFile
Ntfs!NtfsCreateFileLock
...
MpFilter!MpTxfpCreateFileEntryUnsafe
mrxsmb10!MRxSmbFinishLongNameCreateFile
```

```
...
srv!SrvIoCreateFile
```

You can put a sequence of commands in a command file, and use [\\$\\$>< \(Run Script File\)](#) to execute those commands for each function that matches the pattern. Suppose that a file named Commands.txt contains the following commands:

```
dbgcmd

.echo
.echo @#FunctionName
u @#SymbolAddress L1
```

In the following example, the commands in the Commands.txt file are executed for each function, in the PCI module, that matches the pattern *read*.

```
dbgcmd

1: kd> !for_each_function -m:pci -p:*read* -c:$$><Commands.txt

PciReadDeviceConfig
pci!PciReadDeviceConfig [d:\wmm1\drivers\busdrv\pci\config.c @ 349]:
fffff880`00f7b798 48895c2408      mov      qword ptr [rsp+8],rbx

PciReadDeviceSpace
pci!PciReadDeviceSpace [d:\wmm1\drivers\busdrv\pci\config.c @ 1621]:
fffff880`00f7c044 48895c2408      mov      qword ptr [rsp+8],rbx

...
```

See also

[!for_each_module](#)

!for_each_local

Article • 04/03/2024

The `!for_each_local` extension executes a debugger command one time for each local variable in the current frame.

dbgcmd

```
!for_each_local ["CommandString"]
!for_each_local -?
```

Parameters

CommandString

Specifies the debugger commands to execute one time for each local variable in the current stack frame. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands in *CommandString* cannot contain quotation marks.

Within *CommandString*, or within any script that the commands in *CommandString* execute, you can use the `@#Local` alias. This alias is replaced by the name of the local variable. This replacement occurs before *CommandString* is executed and before any other parsing occurs. This alias is case sensitive, and you must add a space before it and add a space after it, even if you enclose the alias in parentheses. If you use C++ expression syntax, you must reference this alias as `@@(@#Local)`.

This alias is available only during the lifetime of the call to `!for_each_local`. Do not confuse this alias with pseudo-registers, fixed-name aliases, or user-named aliases.

-?

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Ext.dll

Additional Information

For more information about how to display and change local variables and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

If you do not specify any arguments, the `!for_each_local` extension lists local variables.

For more information about the local variables, use the [dv \(Display Local Variables\)](#) command.

If you enable verbose debugger output, the display includes the total number of local variables when the extension is called, and every time that *CommandString* is executed for a local variable, that variable and the text of *CommandString* are echoed.

Feedback

Was this page helpful?



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

!for_each_module

Article • 10/25/2023

The `!for_each_module` extension executes a debugger command one time for each loaded module.

dbgcmd

```
!for_each_module ["CommandString"]
!for_each_module -?
```

Parameters

CommandString

Specifies the debugger commands to execute one time for each module in the debugger's module list. If *CommandString* includes multiple commands, you must separate them with semicolons and enclose *CommandString* in quotation marks. If you include multiple commands, the individual commands within *CommandString* cannot contain quotation marks.

You can use the following aliases in *CommandString* or in any script that the commands in *CommandString* executes.

Alias	Data type	Value
@#FileVersion	string	The file version of the module.
@#ProductVersion	string	The product version of the module.
@#ModuleIndex	ULONG	The module number. Modules are enumerated consecutively, starting with zero.
@#ModuleName	string	The module name. This name is typically the file name without the file name extension. In some situations, the module name differs significantly from the file name.
@#ImageName	string	The name of the executable file, including the file name extension. Typically, the full

Alias	Data type	Value
		path is included in user mode but not in kernel mode.
@#LoadedImageName	string	Unless Microsoft CodeView symbols are present, this alias is the same as the image name.
@#MappedImageName	string	In most situations, this alias is NULL . If the debugger is mapping an image file (for example, during minidump debugging), this alias is the name of the mapped image.
@#SymbolFileName	string	The path and name of the symbol file. If you have not loaded any symbols, this alias is the name of the executable file instead.
@#ModuleNameSize	ULONG	The string length of the module name string, plus one.
@#ImageNameSize	ULONG	The string length of the image name string, plus one.
@#LoadedImageNameSize	ULONG	The string length of the loaded image name string, plus one.
@#MappedImageNameSize	ULONG	The string length of the mapped image name string, plus one.
@#SymbolFileNameSize	ULONG	The string length of the symbol file name string, plus one.
@#Base	ULONG64	The address of the start of the image.
@#Size	ULONG	The size of the image, in bytes.
@#End	ULONG64	The address of the end of the image.
@#TimeDateStamp	ULONG	The time and date stamp of the image. If you want to expand this time and date

Alias	Data type	Value
		stamp into a readable date, use the .formats (Show Number Formats) command.
@#Checksum	ULONG	The checksum of the module.
@#Flags	ULONG	The module flags. For a list of the DEBUG_MODULE_Xxx values, see Dbgeng.h.
@#SymbolType	USHORT	The symbol type. For a list of the DEBUG_SYMTYPE_Xxx values, see Dbgeng.h.

These aliases are all replaced before *CommandString* is executed for each module and before any other parsing occurs. These aliases are case sensitive. You must add a space before the alias and a space after it, even if the alias is enclosed in parentheses. If you use C++ expression syntax, you must reference these aliases as @@(@#alias).

These aliases are available only during the lifetime of the call to `!for_each_module`. Do not confuse them with pseudo-registers, fixed-name aliases, or user-named aliases.

-?

Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Additional Information

For more information about how to define and use aliases as shortcuts for entering character strings (including use of the \${ } token), see [Using Aliases](#).

Remarks

If you do not specify any arguments, the `!for_each_module` extension displays general information about the loaded modules. This information is similar to the information that the following command shows.

```
dbgcmd
```

```
!for_each_module .echo @#ModuleIndex : @#Base @#End @#ModuleName @#ImageName  
@#LoadedImageName
```

For more information about loaded and unloaded modules, use the [!m \(List Loaded Modules\)](#) command.

If you enable verbose debugger output, the debugger displays the total number of loaded and unloaded modules when the extension is called, and the debugger displays detailed information about each module (including the values of each available alias) before *CommandString* is executed for that module.

The following examples show how to use the `!for_each_module` extension. The following commands display the global debug flags.

```
dbgcmd
```

```
!for_each_module x ${@#ModuleName}!*Debug*Flag*  
!for_each_module x ${@#ModuleName}!g*Debug*
```

The following command checks for binary corruption in every loaded module, by using the [!chkimg](#) extension:

```
dbgcmd
```

```
!for_each_module !chkimg @#ModuleName
```

The following command searches for the pattern "MZ" in every loaded image.

```
dbgcmd
```

```
!for_each_module s-a @#Base @#End "MZ"
```

The following example demonstrates the use of `@#FileVersion` and `@#ProductVersion` for each module name:

```
dbgcmd
```

```
0:000> !for_each_module .echo @#ModuleName fver = @#FileVersion pver =  
@#ProductVersion  
USER32 fver = 6.0.6000.16438 (vista_gdr.070214-1610) pver = 6.0.6000.16438  
kernel32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386  
ntdll fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386  
notepad fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
```

```
WINSPOOL fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
COMCTL32 fver = 6.10 (vista_rtm.061101-2205) pver = 6.0.6000.16386
SHLWAPI fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
msvcrt fver = 7.0.6000.16386 (vista_rtm.061101-2205) pver = 7.0.6000.16386
GDI32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
RPCRT4 fver = 6.0.6000.16525 (vista_gdr.070716-1600) pver = 6.0.6000.16525
SHELL32 fver = 6.0.6000.16513 (vista_gdr.070626-1505) pver = 6.0.6000.16513
ole32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
ADVAPI32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
COMDLG32 fver = 6.0.6000.16386 (vista_rtm.061101-2205) pver = 6.0.6000.16386
```

!for_each_register

Article • 10/25/2023

The **!for_each_register** extension executes a specified command for each register.

```
dbgcmd
```

```
!for_each_register -c:CommandString  
!for_each_register -?
```

Parameters

-c:CommandString

Specifies the command to be executed for each register. The aliases `@#RegisterName` and `@#RegisterValue` are valid during the execution of the command.

-?

Displays help for the **!for_each_register** extension.

DLL

Ext.dll

Examples

This example lists the name of each register.

```
dbgcmd
```

```
0:000> !for_each_register -c:.echo @#RegisterName  
rax  
rcx  
rdx  
rbx  
...
```

This example executes [!address](#) for each register value.

```
dbgcmd
```

```
0:000> !for_each_register -c:!address ${@#RegisterValue}  
...
```

```
Usage:           Stack
Base Address:   00000008`a568f000
End Address:   00000008`a56a0000
Region Size:   00000000`00011000
State:          00001000    MEM_COMMIT
Protect:        00000004    PAGE_READWRITE
Type:           00020000    MEM_PRIVATE
Allocation Base: 00000008`a5620000
Allocation Protect: 00000004    PAGE_READWRITE
More info:      ~0k
...

```

Remarks

When an alias is an argument to a debugger extension (for example, [!address](#)), use the alias interpreter [\\${} \(Alias Interpreter\)](#) token so that the alias is resolved correctly.

For more information about how to define and use aliases as shortcuts for entering character strings (including use of the [\\${}](#) token), see [Using Aliases](#).

!gflag

Article • 10/25/2023

The **!gflag** extension sets or displays the global flags.

```
dbgcmd  
!gflag [+|-] Value  
!gflag {+|-} Abbreviation  
!gflag -?  
!gflag
```

Parameters

Value

Specifies a 32-bit hexadecimal number. If you do not use a plus sign (+) or minus sign (-), this number becomes the new value of the global flag bit field. If you add a plus sign (+) before this number, the number specifies one or more global flag bits to set to 1. If you add a minus sign (-) before this number, the number specifies one or more global flag bits to set to zero.

Abbreviation

Specifies a single global flag. *Abbreviation* is a three-letter abbreviation for a global flag that is set to 1 (+) or to zero (-).

-?

Displays some Help text for this extension, including a list of global flag abbreviations, in the [Debugger Command window](#).

DLL

Windows 2000	Kdextx86.dll Ntsdexts.dll
Windows XP and later	Exts.dll

Additional Information

You can also set these flags by using the Global Flags utility (Gflags.exe).

Remarks

If you do not specify any arguments, the **!gflag** extension displays the current global flag settings.

The following table contains the abbreviations that you can use for the *Abbreviation* parameter.

Value	Name	Description
0x00000001	"soe"	Stop on exception.
0x00000002	"sls"	Show loader snaps.
0x00000004	"dic"	Debug initial command.
0x00000008	"shg"	Stop if the GUI stops responding (that is, hangs).
0x00000010	"htc"	Enable heap tail checking.
0x00000020	"hfc"	Enable heap free checking.
0x00000040	"hpc"	Enable heap parameter checking.
0x00000080	"hvc"	Enable heap validation on call.
0x00000100	"ptc"	Enable pool tail checking.
0x00000200	"pfc"	Enable pool free checking.
0x00000400	"ptg"	Enable pool tagging.
0x00000800	"htg"	Enable heap tagging.
0x00001000	"ust"	Create a user-mode stack trace DB.
0x00002000	"kst"	Create a kernel-mode stack trace DB.
0x00004000	"otl"	Maintain a list of objects for each type.
0x00008000	"htd"	Enable heap tagging by DLL.
0x00010000	"idp"	Unused.
0x00020000	"d32"	Enable debugging of the Microsoft Win32 subsystem.
0x00040000	"ksl"	Enable loading of kernel debugger symbols.

Value	Name	Description
0x00080000	"dps"	Disable paging of kernel stacks.
0x00100000	"scb"	Enable critical system breaks.
0x00200000	"dhc"	Disable heap coalesce on free.
0x00400000	"ece"	Enable close exception.
0x00800000	"eel"	Enable exception logging.
0x01000000	"eot"	Enable object handle type tagging.
0x02000000	"hpa"	Put heap allocations at the end of pages.
0x04000000	"dwI"	Debug WINLOGON.
0x08000000	"ddp"	Disable kernel-mode DbgPrint and KdPrint output.
0x10000000	NULL	Unused.
0x20000000	"sue"	Stop on unhandled user-mode exception
0x40000000	NULL	Unused.
0x80000000	"dpd"	Disable protected DLL verification.

!gle

Article • 04/03/2024

The **!gle** extension displays the last error value for the current thread.

```
dbgcmd
```

```
!gle [-all]
```

Parameters

-all

Displays the last error for each user-mode thread on the target system. If you omit this parameter in user mode, the debugger displays the last error for the current thread. If you omit this parameter in kernel mode, the debugger displays the last error for the thread that the current [register context](#) specifies.

DLL

Exts.dll

Additional Information

For more information about the [GetLastError](#) routine, see the Windows SDK documentation.

Remarks

The **!gle** extension displays the value of [GetLastError](#) and tries to decode this value.

In kernel mode, the **!gle** extension work only if the debugger can read the thread environment block (TEB).

Feedback

Was this page helpful?

Yes

No

!gs

Article • 04/03/2024

The **!gs** extension analyzes a /GS stack overflow.

```
dbgcmd
```

```
!gs
```

DLL

Ext.dll

Remarks

The **!gsex**tension helps debug buffer overruns. Run **!gs**when you encounter a STATUS_STACK_BUFFER_OVERRUN error, as the following example shows.

```
dbgcmd
```

```
0:000> !gs
Corruption occurred in mshtml!CDoc::OnPaint or one of its callers
Real canary not found at 0x74866010
Canary at gsfailure frame 0x292ea4e7
Corrupted canary 0x0013e2c8: 0x00000000
Corrupted cookie value too generic, skipping init bit-flip check
a caller of mshtml!CDoc::OnPaint has corrupted the EBP from 0x0013e254 to
0x0013
e234
check callers (without canary) of mshtml!CDoc::OnPaint for 0x1 bytes of
overflow
```

The canary doesn't look corrupted. Not sure how we got here
EBP/ESP check skipped: No saved EBP in exception context

Function mshtml!CDoc::OnPaint:

00000000 - 00000004	this	CDoc*
0013de40 - 0013e180	rd	
CDoc::OnPaint::__139::REGION_DAT		
A		
0013e180 - 0013e18c	Lock	CDoc::CLock
0013e18c - 0013e224	DI	CFormDrawInfo
0013e23c - 0013e240	hwndInplace	HWND__*
0013e240 - 0013e244	prc	tagRECT*
0013e248 - 0013e250	ptBefore	tagPOINT
0013e250 - 0013e254	fViewIsReady	int
0013e250 - 0013e254	fHtPalette	int

```
0013e254 - 0013e258 fNoPaint           int
0013e258 - 0013e260 ptAfter            tagPOINT
0013e260 - 0013e264 c                 int
0013e264 - 0013e268 hrgn              HRGN__*
0013e268 - 0013e2a8 ps                tagPAINTSTRUCT
Candidate buffer : ps 0013e268 to 0013e2a7
0013e268 ea 04 01 a7 00 00 00 00-10 01 00 00 f3 00 00 00 .....
0013e278 ed 03 00 00 44 02 00 00-84 e5 13 00 f4 e2 13 00 ....D.....
...
0013e2ac 38 20 01 03 10 e3 13 00-68 6b e6 01 d0 e6 03 00 8 .....hk.....
0013e2bc 80 fa 03 00 0d 00 00 00-10 08 19 00 00 00 00 00 00 .....
0:000>
```

Feedback

Was this page helpful?

 Yes

 No

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

!handle

Article • 10/25/2023

The **!handle** extension displays information about a handle or handles that one or all processes in the target system own.

User-Mode

```
dbgcmd
```

```
!handle [Handle [UMFlags [TypeName]]]  
!handle -?
```

Kernel-Mode

```
dbgcmd
```

```
!handle [Handle [KMFlags [Process [TypeName]]]]
```

Parameters

Handle

Specifies the index of the handle to display. If *Handle* is -1 or if you omit this parameter, the debugger displays data for all handles that are associated with the current process. If *Handle* is 0, the debugger displays data for all handles.

UMFlags

(User mode only) Specifies what the display should contain. This parameter can be a sum of any of the following bit values. (The default value is 0x1.)

Bit 0 (0x1)

Displays handle type information.

Bit 1 (0x2)

Displays basic handle information.

Bit 2 (0x4)

Displays handle name information.

Bit 3 (0x8)

Displays object-specific handle information, when available.

KMFlags

(Kernel mode only) Specifies what the display should contain. This parameter can be a sum of any of the following bit values. (The default value is 0x3.)

Bit 0 (0x1)

Displays basic handle information.

Bit 1 (0x2)

Displays information about objects.

Bit 2 (0x4)

Displays free handle entries. If you do not set this bit and you omit *Handle* or set it to zero, the list of handles that are displayed does not include free handles. If *Handle* specifies a single free handle, it is displayed even if you do not set this bit.

Bit 4 (0x10)

Displays the handle from the kernel handle table instead of the current process.

Bit 5 (0x20)

Interprets the handle as a thread ID or process ID and displays information about the corresponding kernel object.

Process

(Kernel mode only) Specifies a process. You can use the process ID or the hexadecimal address of the process object. This parameter must refer to a currently running process on the target system. If this parameter is -1 or if you omit it, the current process is used. If this parameter is 0, handle information from all processes is displayed.

TypeName

Specifies the type of handle that you want to examine. Only handles that match this type are displayed. *TypeName* is case sensitive. Valid types include Event, Section, File, Port, Directory, SymbolicLink, Mutant, WindowStation, Semaphore, Key, Token, Process, Thread, Desktop, IoCompletion, Timer, Job, and WaitablePort.

-?

(User mode only) Displays some Help text for this extension in the [Debugger Command window](#).

DLL

Windows 2000	Kdextx86.dll Uext.dll Ntsdexts.dll
Windows XP and later	Kdexts.dll Uext.dll Ntsdexts.dll

Additional Information

For more information about handles, see the [!htrace](#) extension, the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

You can use the [!handle](#) extension during user-mode and kernel-mode live debugging. You can also use this extension on kernel-mode dump files. However, you cannot use this extension on user-mode dump files, unless you specifically created them with handle information. (You can create such dump files by using the [.dump /mh \(Create Dump File\)](#) command.)

During live user-mode debugging, you can use the [.closehandle \(Close Handle\)](#) command to close one or more handles.

The following examples are user-mode examples of the [!handle](#) extension. The following command displays a list of all handles.

```
dbgcmd

0:000> !handle
Handle 4
  Type      Section
Handle 8
  Type      Event
Handle c
  Type      Event
Handle 10
  Type      Event
Handle 14
  Type      Directory
Handle 5c
  Type      File
6 Handles
Type      Count
Event     3
Section   1
File      1
Directory 1
```

The following command displays detailed information about handle 0x8.

```
dbgcmd
```

```
0:000> !handle 8 f
Handle 8
  Type          Event
  Attributes    0
  GrantedAccess 0x100003:
    Synch
    QueryState,ModifyState
  HandleCount   2
  PointerCount  3
  Name          <none>
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting
```

The following examples are kernel-mode examples of `!handle`. The following command lists all handles, including free handles.

```
dbgcmd

kd> !handle 0 4
processor number 0
PROCESS 80559800 SessionId: 0 Cid: 0000 Peb: 00000000 ParentCid: 0000
  DirBase: 00039000 ObjectTable: e1000d60 TableSize: 380.
  Image: Idle

New version of handle table at e1002000 with 380 Entries in use

  0000: free handle, Entry address e1002000, Next Entry ffffffff
  0004: Object: 80ed5238 GrantedAccess: 001f0fff
  0008: Object: 80ed46b8 GrantedAccess: 00000000
  000c: Object: e1281d00 GrantedAccess: 000f003f
  0010: Object: e1013658 GrantedAccess: 00000000
  .....
  0168: Object: ffb6c748 GrantedAccess: 00000003 (Protected)
  016c: Object: ff811f90 GrantedAccess: 0012008b
  0170: free handle, Entry address e10022e0, Next Entry 00000458
  0174: Object: 80dfd5c8 GrantedAccess: 001f01ff
  .....
```

The following command show detailed information about handle 0x14 in the kernel handle table.

```
dbgcmd

kd> !handle 14 13
processor number 0
PROCESS 80559800 SessionId: 0 Cid: 0000 Peb: 00000000 ParentCid: 0000
  DirBase: 00039000 ObjectTable: e1000d60 TableSize: 380.
  Image: Idle
```

```
Kernel New version of handle table at e1002000 with 380 Entries in use
0014: Object: e12751d0 GrantedAccess: 0002001f
Object: e12751d0 Type: (80ec8db8) Key
    ObjectHeader: e12751b8
        HandleCount: 1 PointerCount: 1
        Directory Object: 00000000 Name:
\REGISTRY\MACHINE\SYSTEM\CONTROLSET001\CONTROL\SESSION MANAGER\EXECUTIVE
```

The following command shows information about all handles to Section objects in all processes.

```
dbgcmd

!handle 0 3 0 Section
...
PROCESS ffffffa8004f48940
    SessionId: none Cid: 0138 Peb: 7f6639bf000 ParentCid: 0004
    DirBase: 10cb74000 ObjectTable: fffff8a00066f700 HandleCount: 39.
    Image: smss.exe

Handle table at fffff8a00066f700 with 39 entries in use

0040: Object: fffff8a000633f00 GrantedAccess: 00000006 (Inherit) Entry:
fffff8a000670100
Object: fffff8a000633f00 Type: (fffffa80035fef20) Section
    ObjectHeader: fffff8a000633ed0 (new version)
        HandleCount: 1 PointerCount: 262144
...
```

!heap

Article • 10/25/2023

The **!heap** extension displays heap usage information, controls breakpoints in the heap manager, detects leaked heap blocks, searches for heap blocks, or displays page heap information.

This extension supports the segment heap and the NT heap. Use **!heap** with no parameter to list all heaps and their type.

dbgcmd

```
!heap [HeapOptions] [ValidationOptions] [Heap]
!heap -b [{alloc|realloc|free} [Tag]] [Heap | BreakAddress]
!heap -B {alloc|realloc|free} [Heap | BreakAddress]
!heap -l
!heap -s [SummaryOptions] [StatHeapAddress]
!heap -i HeapAddress
!heap -x [-v] Address
!heap -p [PageHeapOptions]
!heap -srch [Size] Pattern
!heap -flt FilterOptions
!heap -stat [-h Handle [-grp GroupBy [MaxDisplay]]]
!heap [-p] -?
!heap -triage [Handle | Address]
```

Segment and NT Heap Parameters

These parameters work with Segment and NT heaps.

-s

Specifies that summary information is being requested. If *SummaryOptions* and *StatHeapAddress* are omitted, then summary information is displayed for all heaps associated with the current process.

SummaryOptions

Can be any combination of the following options. The *SummaryOptions* are not case-sensitive. Type **!heap -s -?** for additional information.

Option	Effect
-v	Verifies all data blocks.
-b BucketSize	Specifies the bucket size. The default is 1024

Option	Effect
	bits.
-d <i>DumpBlockSize</i>	Specifies the bucket size.
-a	Dumps all heap blocks.
-c	Specifies that the contents of each block should be displayed.

-triage [*Handle* | *Address*]

Causes the debugger to automatically search for failures in a process's heaps. If a heap handle is specified as an argument, that heap is examined; otherwise, all the heaps are searched for one that contains the given address, and if one is found, it is examined.

Using **-triage** is the only way to validate low-fragmentation heap (LFH) corruption.

-x [-v]

Causes the debugger to search for the heap block containing the specified address. If **-v** is added, the command will search the entire virtual memory space of the current process for pointers to this heap block.

-l

Causes the debugger to detect leaked heap blocks.

-i *Address* -h *HeapAddress*

Displays information about the specified *Heap*.

Address

Specifies the address to search for.

-?

Displays some brief Help text for this extension in the Debugger Command window. Use **!heap -?** for generic help, and **!heap -p -?** for page heap help.

NT Heap Parameters

These parameters work only with the NT Heap.

HeapOptions

Can be any combination of the following options. The *HeapOptions* values are case-sensitive.

Option	Effect
-v	<p>Causes the debugger to validate the specified heap.</p> <p>Note This option does not detect low fragmentation heap (LFH) corruption. Use -triage instead.</p>
-a	<p>Causes the display to include all information for the specified heap. Size, in this case, is rounded up to the heap granularity. (Running !heap with the -a option is equivalent to running it with the three options -h -f -m, which can take a long time.)</p>
-h	<p>Causes the display to include all non-LFH entries for the specified heap.</p>
-hl	<p>Causes the display to include all the entries for the specified heap(s), including LFH entries.</p>
-f	<p>Causes the display to include all the free list entries for the specified heap.</p>
-m	<p>Causes the display to include all the segment entries for the specified heap.</p>
-t	<p>Causes the display to include the tag information for the specified heap.</p>
-T	<p>Causes the display to include the pseudo-tag entries for the specified heap.</p>
-g	<p>Causes the display to include the global tag information. Global tags are associated with each untagged allocation.</p>
-s	<p>Causes the display to include summary information for the specified heap.</p>
-k	<p>(x86-based targets only) Causes the display to include the stack backtrace associated with each entry.</p>

ValidationOptions

Can be any single one of the following options. The *ValidationOptions* are case-sensitive.

Option	Effect
-D	Disables validate-on-call for the specified heap.
-E	Enables validate-on-call for the specified heap.
-d	Disables heap checking for the specified heap.
-e	Enables heap checking for the specified heap.

-i *Heap Address or HeapAddress*

Displays information about the specified *Heap*.

BreakAddress

Specifies the address of a block where a breakpoint is to be set or removed.

-b

Causes the debugger to create a conditional breakpoint in the heap manager. The **-b** option can be followed by **alloc**, **realloc**, or **free**; this specifies whether the breakpoint will be activated by allocating, reallocating, or freeing memory. If *BreakAddress* is used to specify the address of the block, the breakpoint type can be omitted. If *Heap* is used to specify the heap address or heap index, the type must be included, as well as the *Tag* parameter.

Tag

Specifies the tag name within the heap.

-B

Causes the debugger to remove a conditional breakpoint from the heap manager. The breakpoint type (**alloc**, **realloc**, or **free**) must be specified, and must be the same as that used with the **-b** option.

StatHeapAddress

Specifies the address of the heap. If this is 0 or omitted, all heaps associated with the current process are displayed.

-p

Specifies that page heap information is being requested. If this is used without any *PageHeapOptions*, all page heaps will be displayed.

PageHeapOptions

Can be any single one of the following options. The *PageHeapOptions* are case-sensitive. If no options are specified, then all possible page heap handles will be displayed.

Option	Effect
-h Handle	Causes the debugger to display detailed information about a page heap with handle <i>Handle</i> .
-a Address	Causes the debugger to find the page heap whose block contains <i>Address</i> . Full details of how this address relates to full-page heap blocks will be included, such as whether this address is part of a page heap, its offset inside the block, and whether the block is allocated or was freed. Stack traces are included whenever available. When using this option, size is displayed in multiples of the heap allocation granularity.
-t[c s] [Traces]	Causes the debugger to display the collected traces of the heavy heap users. <i>Traces</i> specifies the number of traces to display; the default is four. If there are more traces than the specified number, the earliest traces are displayed. If -t or -tc is used, the traces are sorted by count usage. If -ts is used, the traces are sorted by size. (The -tc and -ts options are supported in Windows XP only; the -t option is supported only in Windows XP and earlier versions of Windows.)
-fi [Traces]	Causes the debugger to display the most recent fault injection traces. <i>Traces</i> specifies the quantity to be displayed; the default is 4.
-all	Causes the debugger to display detailed information about all page heaps.
-?	Causes the debugger to display page heap help, including a diagram of heap blocks. (These diagrams can also be seen in the following Remarks section.)

Before you can use any **!heap -p** extension command, the page heap must be enabled for your target process. See details in the following Remarks section.

-srch

Scans all heaps for the given pattern.

Pattern

Specifies a pattern for which to look.

Size

Can be any single one of the following options. This specifies the size of the pattern. The '-' is required.

Option	Effect
-b	The pattern is one BYTE in size.
-w	The pattern is one WORD in size.
-d	The pattern is one DWORD in size.
-q	The pattern is one QWORD in size.

If none of the above are specified, then the pattern is assumed to be of the same size as the machine pointer.

-flt

Limits the display to include only heaps with the specified size or size range.

FilterOptions

Can be any single one of the following options. The *FilterOptions* are case-sensitive.

Option	Effect
s <i>Size</i>	Limits the display to include only heaps of the specified size.
r <i>SizeMin SizeMax</i>	Limits the display to include only heaps within the specified size range.

-stat

Displays usage statistics for the specified heap.

-h *Handle*

Causes usage statistics for only the heap at *Handle* to be displayed. If *Handle* is 0 or omitted, then usage statistics for all heaps are displayed.

-grp *GroupBy*

Reorders the display as specified by *GroupBy*. The options for *GroupBy* can be found in the following table.

Option	Effect
A	Displays the usage statistics according to allocation size.

Option	Effect
B	Displays the usage statistics according to block count.
S	Displays the usage statistics according to the total size of each allocation.

MaxDisplay

Limits the output to only *MaxDisplay* number of lines.

DLL

Windows XP and later	Ext.dll Exts.dll
----------------------	------------------

Additional Information

For information about heaps, see the following resources:

Book: *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

[Example 11: Enabling Page Heap Verification](#)

[Example 12: Using Page Heap Verification to Find a Bug](#)

For information on using the Heap memory process logger, see [Example 11: Starting a Private Trace Session](#)

Remarks

This extension command can be used to perform a variety of tasks.

The standard **!heap** command is used to display heap information for the current process. (This should be used only for user-mode processes. The **!pool** extension command should be used for system processes.)

The **!heap -b** and **!heap -B** commands are used to create and delete conditional breakpoints in the heap manager.

The **!heap -l** command detects leaked heap blocks. It uses a garbage collector algorithm to detect all busy blocks from the heaps that are not referenced anywhere in the process address space. For huge applications, it can take a few minutes to complete. This command is only available in Windows XP and later versions of Windows.

The **!heap -x** command searches for a heap block containing a given address. If the **-v** option is used, this command will additionally search the entire virtual memory space of the current process for pointers to this heap block. This command is only available in Windows XP and later versions of Windows.

The **!heap -p** command displays various forms of page heap information. Before using **!heap -p**, you must enable the page heap for the target process. This is done through the Global Flags (gflags.exe) utility. To do this, start the utility, fill in the name of the target application in the **Image File Name** text box, select **Image File Options** and **Enable page heap**, and select **Apply**. Alternatively, you can start the Global Flags utility from a Command Prompt window by typing **gflags /i xxx.exe +hpa**, where **xxx.exe** is the name of the target application.

The **!heap -p -t[c|s]** commands are not supported beyond Windows XP. Use the [UMDH](#) tool provided with the debugger package to obtain similar results.

The **!heap -srch** command displays those heap entries that contain a certain specified pattern.

The **!heap -flt** command limits the display to only heap allocations of a specified size.

The **!heap -stat** command displays heap usage statistics.

Here is an example of the standard **!heap** command:

```
dbgcmd

0:000> !ntsdexts.heap -a
Index   Address   Name       Debugging options enabled
1:   00250000
      Segment at 00250000 to 00350000 (00056000 bytes committed)
      Flags:           50000062
      ForceFlags:      40000060
      Granularity:    8 bytes
      Segment Reserve: 00100000
      Segment Commit:  00004000
      DeCommit Block Thres:00000400
      DeCommit Total Thres:00002000
      Total Free Size: 000003be
      Max. Allocation Size:7fffffff
      Lock Variable at: 00250b54
      Next TagIndex:   0012
      Maximum TagIndex: 07ff
      Tag Entries:     00350000
      PsuedoTag Entries: 00250548
      Virtual Alloc List: 00250050
      UCR FreeList:    002504d8
      128-bit bitmap of free lists
      FreeList Usage:   00000014 00000000 00000000 00000000
```

```

          Free   Free
          List   List
#      Head     Blink    Flink
FreeList[ 00 ] at 002500b8: 002a4378 . 002a4378
                           0x02 - HEAP_ENTRY_EXTRA_PRESENT
                           0x04 - HEAP_ENTRY_FILL_PATTERN
Entry      Prev     Cur     0x10 - HEAP_ENTRY_LAST_ENTRY

Address  Size   Size   flags
002a4370: 00098 . 01c90 [14] - free
FreeList[ 02 ] at 002500c8: 0025cb30 . 002527b8
002527b0: 00058 . 00010 [04] - free
0025cb28: 00088 . 00010 [04] - free
FreeList[ 04 ] at 002500d8: 00269a08 . 0026e530
0026e528: 00038 . 00020 [04] - free
0026a4d0: 00038 . 00020 [06] - free
0026f9b8: 00038 . 00020 [04] - free
0025cda0: 00030 . 00020 [06] - free
00272660: 00038 . 00020 [04] - free
0026ab60: 00038 . 00020 [06] - free
00269f20: 00038 . 00020 [06] - free
00299818: 00038 . 00020 [04] - free
0026c028: 00038 . 00020 [06] - free
00269a00: 00038 . 00020 [46] - free

Segment00 at 00250b90:
Flags:          00000000
Base:           00250000
First Entry:   00250bc8
Last Entry:    00350000
Total Pages:   00000080
Total UnCommit: 00000055
Largest UnCommit: 000aa000
UnCommitted Ranges: (1)
002a6000: 000aa000

Heap entries for Segment00 in Heap 250000
                           0x01 - HEAP_ENTRY_BUSY
                           0x02 - HEAP_ENTRY_EXTRA_PRESENT
                           0x04 - HEAP_ENTRY_FILL_PATTERN
                           0x08 - HEAP_ENTRY_VIRTUAL_ALLOC
                           0x10 - HEAP_ENTRY_LAST_ENTRY
                           0x20 - HEAP_ENTRY_SETTABLE_FLAG1
                           0x40 - HEAP_ENTRY_SETTABLE_FLAG2
Entry      Prev     Cur     0x80 - HEAP_ENTRY_SETTABLE_FLAG3

Address  Size   Size   flags       (Bytes used)      (Tag name)
00250000: 00000 . 00b90 [01] - busy (b90)
00250b90: 00b90 . 00038 [01] - busy (38)
00250bc8: 00038 . 00040 [07] - busy (24), tail fill (NTDLL!LDR Database)
00250c08: 00040 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00250c68: 00060 . 00028 [07] - busy (10), tail fill (NTDLL!LDR Database)
00250c90: 00028 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00250cf0: 00060 . 00050 [07] - busy (38), tail fill (Objects= 80)
00250d40: 00050 . 00048 [07] - busy (2e), tail fill (NTDLL!LDR Database)

```

```

00250d88: 00048 . 00c10 [07] - busy (bf4), tail fill (Objects>1024)
00251998: 00c10 . 00030 [07] - busy (12), tail fill (NTDLL!LDR Database)
...
002525c0: 00030 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
00252620: 00060 . 00050 [07] - busy (38), tail fill (NTDLL!LDR Database)
00252670: 00050 . 00040 [07] - busy (22), tail fill (NTDLL!CSRSS Client)
002526b0: 00040 . 00040 [07] - busy (24), tail fill (Objects= 64)
002526f0: 00040 . 00040 [07] - busy (24), tail fill (Objects= 64)
00252730: 00040 . 00028 [07] - busy (10), tail fill (Objects= 40)
00252758: 00028 . 00058 [07] - busy (3c), tail fill (Objects= 88)
002527b0: 00058 . 00010 [04] free fill
002527c0: 00010 . 00058 [07] - busy (3c), tail fill (NTDLL!LDR Database)
00252818: 00058 . 002d0 [07] - busy (2b8), tail fill (Objects= 720)
00252ae8: 002d0 . 00330 [07] - busy (314), tail fill (Objects= 816)
00252e18: 00330 . 00330 [07] - busy (314), tail fill (Objects= 816)
00253148: 00330 . 002a8 [07] - busy (28c), tail fill (NTDLL!LocalAtom)
002533f0: 002a8 . 00030 [07] - busy (18), tail fill (NTDLL!LocalAtom)
00253420: 00030 . 00030 [07] - busy (18), tail fill (NTDLL!LocalAtom)
00253450: 00030 . 00098 [07] - busy (7c), tail fill (BASEDLL!LMEM)
002534e8: 00098 . 00060 [07] - busy (44), tail fill (BASEDLL!TMP)
00253548: 00060 . 00020 [07] - busy (1), tail fill (Objects= 32)
00253568: 00020 . 00028 [07] - busy (10), tail fill (Objects= 40)
00253590: 00028 . 00030 [07] - busy (16), tail fill (Objects= 48)
...
0025ccb8: 00038 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
0025cd18: 00060 . 00058 [07] - busy (3c), tail fill (NTDLL!LDR Database)
0025cd70: 00058 . 00030 [07] - busy (18), tail fill (NTDLL!LDR Database)
0025cda0: 00030 . 00020 [06] free fill (NTDLL!Temporary)
0025cdc0: 00020 . 00258 [07] - busy (23c), tail fill (Objects= 600)
0025d018: 00258 . 01018 [07] - busy (1000), tail fill (Objects>1024)
0025e030: 01018 . 00060 [07] - busy (48), tail fill (NTDLL!LDR Database)
...
002a4190: 00028 . 00118 [07] - busy (100), tail fill (BASEDLL!GMEM)
002a42a8: 00118 . 00030 [07] - busy (18), tail fill (Objects= 48)
002a42d8: 00030 . 00098 [07] - busy (7c), tail fill (Objects= 152)
002a4370: 00098 . 01c90 [14] free fill
002a6000:      000aa000      - uncommitted bytes.

```

Here is an example of the !heap -l command:

```

dbgcmd

1:0:011> !heap -l
1:Heap 00170000
Heap 00280000
Heap 00520000
Heap 00b50000
Heap 00c60000
Heap 01420000
Heap 01550000
Heap 016d0000
Heap 019b0000
Heap 01b40000

```

Scanning VM ...							
##	Entry	User	Heap	Segment	Size	PrevSize	Flags
001b2958	001b2960	00170000	00000000		40	18	busy extra
001b9cb0	001b9cb8	00170000	00000000		80	300	busy extra
001ba208	001ba210	00170000	00000000		80	78	busy extra
001cbc90	001cbc98	00170000	00000000		e0	48	busy extra
001cbd70	001cbd78	00170000	00000000		d8	e0	busy extra
001cbe90	001cbe98	00170000	00000000		68	48	busy extra
001cbef8	001cbf00	00170000	00000000		58	68	busy extra
001cc078	001cc080	00170000	00000000		f8	128	busy extra
001cc360	001cc368	00170000	00000000		80	50	busy extra
001cc3e0	001cc3e8	00170000	00000000		58	80	busy extra
001fe550	001fe558	00170000	00000000		150	278	busy extra
001fe6e8	001fe6f0	00170000	00000000		48	48	busy extra
002057a8	002057b0	00170000	00000000		58	58	busy extra
00205800	00205808	00170000	00000000		48	58	busy extra
002058b8	002058c0	00170000	00000000		58	70	busy extra
00205910	00205918	00170000	00000000		48	58	busy extra
00205958	00205960	00170000	00000000		90	48	busy extra
00246970	00246978	00170000	00000000		60	88	busy extra
00251168	00251170	00170000	00000000		78	d0	busy extra
user_flag							
00527730	00527738	00520000	00000000		40	40	busy extra
00527920	00527928	00520000	00000000		40	80	busy extra
21 leaks detected.							

The table in this example contains all 21 leaks found.

Here is an example of the !heap -x command:

dbgcmd							
0:011>	!heap 002057b8 -x						
##	Entry	User	Heap	Segment	Size	PrevSize	Flags
002057a8	002057b0	00170000	00170640		58	58	busy extra

Here is an example of the !heap -x -v command:

dbgcmd							
1:0:011>	!heap 002057b8 -x -v						
##	1:Entry	User	Heap	Segment	Size	PrevSize	Flags
002057a8	002057b0	00170000	00170640		58	58	busy extra
Search VM for address range 002057a8 - 002057ff : 00205990 (002057d0),							

In this example, there is a pointer to this heap block at address 0x00205990.

Here is an example of the !heap -flt s command:

```
dbgcmd  
0:001>!heap -flt s 0x50
```

This will display all of the allocations of size 0x50.

Here is an example of the !heap -flt r command:

```
dbgcmd  
0:001>!heap -flt r 0x50 0x80
```

This will display each allocation whose size is between 0x50 and 0x7F.

Here is an example of the !heap -srch command.

```
dbgcmd  
0:001> !heap -srch 77176934  
_HEAP @ 00090000  
in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state  
    00099A48: 0018 : 0005 [01] - 00099A50 (000000B8) - (busy)  
        ole32!CALLFRAME_CACHE<INTERFACE_HELPER_CLSID>::`vftable'  
_HEAP @ 00090000  
in HEAP_ENTRY: Size : Prev Flags - UserPtr UserSize - state  
    00099B58: 0018 : 0005 [01] - 00099B60 (000000B8) - (busy)  
        ole32!CALLFRAME_CACHE<INTERFACE_HELPER_CLSID>::`vftable'
```

The following diagrams show the arrangement of heap blocks.

Light page heap block -- allocated:

```
dbgcmd  
+-----+-----+---+  
|       |           |   |  
+-----+-----+---+  
^       ^           ^  
|       |           8 suffix bytes (filled with 0xA0)  
|       User allocation (filled with E0 if zeroing not requested)  
Block header (starts with 0xABCDAAAA and ends with 0xDCBAAAAA)
```

Light page heap block -- freed:

```
dbgcmd
```

```
+-----+-----+---+
|     |           |   |
+-----+-----+---+
^       ^       ^
|           |       8 suffix bytes (filled with 0xA0)
|           User allocation (filled with F0 bytes)
Block header (starts with 0xABCDAAA9 and ends with 0xDCBAAA9)
```

Full page heap block -- allocated:

```
dbgcmd
```

```
+-----+-----+-----+
|     |           |   | ... N/A page
+-----+-----+-----+
^       ^       ^
|           |       0-7 suffix bytes (filled with 0xD0)
|           User allocation (if zeroing not requested, filled
|             with C0)
Block header (starts with 0xABCDBBBB and ends with 0xDCBABBBA)
```

Full page heap block -- freed:

```
dbgcmd
```

```
+-----+-----+-----+
|     |           |   | ... N/A page
+-----+-----+-----+
^       ^       ^
|           |       0-7 suffix bytes (filled with 0xD0)
|           User allocation (filled with F0 bytes)
Block header (starts with 0xABCDBBAA and ends with 0xDCBABBAA)
```

To see the stack trace of the allocation or the freeing of a heap block or full page heap block, use [dt DPH_BLOCK_INFORMATION](#) with the header address, followed by [dds](#) with the block's **StackTrace** field.

!help

Article • 10/25/2023

The **!help** extension displays help text that describes the extension commands exported from the extension DLL.

Do not confuse this extension command with the [? \(Command Help\)](#) or [.help \(Meta-Command Help\)](#) commands.

```
dbgcmd  
![ExtensionDLL.]help [-v] [CommandName]
```

Parameters

ExtensionDLL

Displays help for the specified extension DLL. Type the name of an extension DLL without the .dll file name extension. If the DLL file is not in the extension search path (as displayed by using [.chain \(List Debugger Extensions\)](#)), include the path to the DLL file. For example, to display help for uext.dll, type **!uext.help** or **!Path\winext\uext.help**.

If you omit the *ExtensionDLL*, the debugger will display the help text for the first extension DLL in the list of loaded DLLs.

-v

Displays the most detailed help text available. This feature is not supported in all DLLs.

CommandName

Displays only the help text for the specified command. This feature is not supported in all DLLs or for all commands.

DLL

This extension is supported by most extension DLLs.

Remarks

Some individual commands will also display a help text if you use the **/?** or **-?** parameter with the command name.

!homedir

Article • 04/03/2024

The **!homedir** extension sets the default directory used by the symbol server and the source server.

dbgcmd

```
!homedir Directory  
!homedir
```

Parameters

Directory

Specifies the new directory to use as the home directory.

DLL

Dbghelp.dll

Remarks

If the **!homedir** extension is used with no argument, the current home directory is displayed.

The cache for a source server is located in the `src` subdirectory of the home directory. The downstream store for a symbol server defaults to the `sym` subdirectory of the home directory, unless a different location is specified.

When WinDbg is started, the home directory is the directory where Debugging Tools for Windows was installed. The **!homedir** extension can be used to change this value.

Feedback

Was this page helpful?

 Yes

 No

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

!hstring

Article • 10/25/2023

The **!hstring** extension displays the fields of an **HSTRING**. The last item in the display is the string itself.

dbgcmd

!hstring Address

Parameters

Address

The address of an **HSTRING**.

Remarks

The **HSTRING** data type supports strings that have embedded NULL characters. However, the **!hstring** extension displays the string only up to the first NULL character. To see the entire string including the embedded NULL characters, use the [!hstring2](#) extension.

See also

[Windows Runtime Debugger Commands](#)

[!hstring2](#)

[!winrterr](#)

!hstring2

Article • 10/25/2023

The **!hstring2** extension displays an entire **HSTRING** including any embedded NULL characters in the string itself.

```
dbgcmd
```

```
!hstring2 Address
```

Parameters

Address

The address of an **HSTRING**.

See also

[Windows Runtime Debugger Commands](#)

[!hstring](#)

[!winrterr](#)

!htrace

Article • 10/25/2023

The **!htrace** extension displays stack trace information for one or more handles.

User-Mode Syntax

```
dbgcmd

!htrace [Handle [Max_Traces]]
!htrace -enable [Max_Traces]
!htrace -snapshot
!htrace -diff
!htrace -disable
!htrace -?
```

Kernel-Mode Syntax

```
dbgcmd

!htrace [Handle [Process [Max_Traces]]]
!htrace -?
```

Parameters

Handle Specifies the handle whose stack trace will be displayed. If *Handle* is 0 or omitted, stack traces for all handles in the process will be displayed.

Process (Kernel mode only) Specifies the process whose handles will be displayed. If *Process* is 0 or omitted, then the current process is used. In user mode, the current process is always used.

Max_Traces Specifies the maximum number of stack traces to display. In user mode, if this parameter is omitted, then all the stack traces for the target process will be displayed.

-enable (User mode only) Enables handle tracing and takes the first snapshot of the handle information to use as the initial state by the **-diff** option.

-snapshot (User mode only) Takes a snapshot of the current handle information to use as the initial state by the **-diff** option.

-diff (User mode only) Compares current handle information with the last snapshot of handle information that was taken. Displays all handles that are still open.

-disable (User mode only) Disables handle tracing.

-?

Displays brief help text for this extension in the Debugger Command window.

DLL

Windows XP and later

Kdexts.dll Ntsdexts.dll

Additional Information

To display further information about a specific handle, use the [!handle](#) extension.

For information about handles, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Before **!htrace** can be used, handle tracing must be enabled. One way to enable handle tracing is to enter the **!htrace -enable** command. When handle tracing is enabled, stack trace information is saved each time the process opens a handle, closes a handle, or references an invalid handle. It is this stack trace information that **!htrace** displays.

Note You can also enable handle tracing by activating Application Verifier for the target process and selecting the **Handles** option.

Some of the traces reported by **!htrace** may be from a different process context. In this case, the return addresses may not resolve properly in the current process context, or may resolve to the wrong symbols.

The following example displays information about all handles in process 0x81400300.

```
dbgcmd
```

```
kd> !htrace 0 81400300
Process 0x81400300
ObjectTable 0xE10CCF60
##
```

```
Handle 0x7CC - CLOSE:  
0x8018FCB9: ntoskrnl!ExDestroyHandle+0x103  
0x801E1D12: ntoskrnl!ObpCloseHandleTableEntry+0xE4  
0x801E1DD9: ntoskrnl!ObpCloseHandle+0x85  
0x801E1EDD: ntoskrnl!NtClose+0x19  
0x010012C1: badhandle!mainCRTStartup+0xE3  
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D  
  
Handle 0x7CC - OPEN:  
0x8018F44A: ntoskrnl!ExCreateHandle+0x94  
0x801E3390: ntoskrnl!ObpCreateUnnamedHandle+0x10C  
0x801E7317: ntoskrnl!ObInsertObject+0xC3  
0x77DE23B2: KERNEL32!CreateSemaphoreA+0x66  
0x010011C5: badhandle!main+0x45  
0x010012C1: badhandle!mainCRTStartup+0xE3  
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D  
  
Handle 0x7DC - BAD REFERENCE:  
0x8018F709: ntoskrnl!ExMapHandleToPointerEx+0xEA  
0x801E10F2: ntoskrnl!ObReferenceObjectByHandle+0x12C  
0x801902BE: ntoskrnl!NtSetEvent+0x6C  
0x80154965: ntoskrnl!_KiSystemService+0xC4  
0x010012C1: badhandle!mainCRTStartup+0xE3  
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D  
  
Handle 0x7DC - CLOSE:  
0x8018FCB9: ntoskrnl!ExDestroyHandle+0x103  
0x801E1D12: ntoskrnl!ObpCloseHandleTableEntry+0xE4  
0x801E1DD9: ntoskrnl!ObpCloseHandle+0x85  
0x801E1EDD: ntoskrnl!NtClose+0x19  
0x010012C1: badhandle!mainCRTStartup+0xE3  
## 0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D  
  
Handle 0x7DC - OPEN:  
0x8018F44A: ntoskrnl!ExCreateHandle+0x94  
0x801E3390: ntoskrnl!ObpCreateUnnamedHandle+0x10C  
0x801E7317: ntoskrnl!ObInsertObject+0xC3  
0x77DE265C: KERNEL32!CreateEventA+0x66  
0x010011A0: badhandle!main+0x20  
0x010012C1: badhandle!mainCRTStartup+0xE3  
0x77DE0B2F: KERNEL32!BaseProcessStart+0x3D  
##  
  
Parsed 0x6 stack traces.  
Dumped 0x5 stack traces.
```

!imggp

Article • 04/03/2024

The **!imggp** extension displays the global pointer (GP) directory entry value for a 64-bit image.

dbgcmd

!imggp Address

Parameters

Address

Specifies the base address of the image.

DLL

Ext.dll

Feedback

Was this page helpful?

 Yes

 No

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

!imgreloc

Article • 04/03/2024

The **!imgreloc** extension displays the addresses of each loaded module and indicates their former addresses before they were relocated.

```
dbgcmd
```

```
!imgreloc Address
```

Parameters

Address

Specifies the base address of the image.

DLL

Ext.dll

Remarks

Here is an example:

```
dbgcmd
```

```
0:000> !imgreloc 00400000
00400000 Prymes - at preferred address
010e0000 appvcore - RELOCATED from 00400000
5b2f0000 verifier - at preferred address
5d160000 ShimEng - at preferred address
```

Feedback

Was this page helpful?

 Yes

 No

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

!kuser

Article • 04/03/2024

The **!kuser** extension displays the shared user-mode page (KUSER_SHARED_DATA).

```
dbgcmd
```

```
!kuser
```

DLL

Exts.dll

Remarks

The KUSER_SHARED_DATA page gives resource and other information about the user who is currently logged on.

Here is an example. Note that, in this example, the tick count is displayed in both its raw form and in a more user-friendly form, which is in parentheses. The user-friendly display is available only in Windows XP and later.

```
dbgcmd
```

```
kd> !kuser
_KUSER_SHARED_DATA at 7ffe0000
TickCount:    fa00000 * 00482006 (0:20:30:56.093)
TimeZone Id: 2
ImageNumber Range: [14c .. 14c]
Crypto Exponent: 0
SystemRoot: 'F:\WINDOWS'
```

Feedback

Was this page helpful?

 Yes

 No

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

!list

Article • 10/25/2023

The **!list** extension executes the specified debugger commands repeatedly, once for every element in a linked list.

dbgcmd

```
!list -t [Module!]Type.Field -x "Commands" [-a "Arguments"] [Options]
StartAddress
!list " -t [Module!]Type.Field -x \"Commands\" [-a \"Arguments\"] [Options]
StartAddress "
!list -h
```

Parameters

Module

An optional parameter specifying the module that defines this structure. If there is a chance that *Type* may match a valid symbol in a different module, you should include *Module* to eliminate the ambiguity.

Type

Specifies the name of a data structure.

Field

Specifies the field containing the list link. This can actually be a sequence of fields separated by periods (in other words, *Type*.*Field*.*Subfield*.*Subsubfield*, and so on).

-x "Commands"

Specifies the commands to execute. This can be any combination of debugger commands. It must be enclosed in quotation marks. If multiple commands are specified, separate them with semicolons, enclose the entire collection of **!list** arguments in quotation marks, and use an escape character (\) before each quotation mark inside these outer quotation marks. If *Commands* is omitted, the default is [dp \(Display Memory\)](#).

-a "Arguments"

Specifies the arguments to pass to the *Commands* parameter. This must be enclosed in quotation marks. *Arguments* can be any valid argument string that would normally be allowed to follow this command, except that *Arguments* cannot contain quotation marks. If the pseudo-register `$extret` is included in *Commands*, the *-a "Arguments"* parameter can be omitted.

Options Can be any number of the following options:

-e

Echoes the command being executed for each element.

-m Max

Specifies the maximum number of elements to execute the command for.

StartAddress

Specifies the address of the first data structure. This is the address at the top of the structure, not necessarily the address of the link field.

-h

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows 2000	Ext.dll
Windows XP and later	Ext.dll

Remarks

The **!list** extension will go through the linked list and issue the specified command once for each list element.

The pseudo-register **\$extret** is set to the value of the list-entry address for each list element. For each element, the command string *Commands* is executed. This command string can reference this pseudo-register using the **\$extret** syntax. If this does not appear in the command string, the value of the list-entry address is appended to the end of the command string before execution. If you need to specify where this value should appear in your command, you must specify this pseudo-register explicitly.

This command sequence will run until the list terminates in a null pointer, or terminates by looping back onto the first element. If the list loops back onto a later element, this command will not stop. However, you can stop this command at any time by using **CTRL+C** in KD and CDB, or **Debug | Break** or **CTRL+BREAK** in WinDbg.

Each time a command is executed, the address of the current structure will be used as the *default address* if the command being used has optional address parameters.

Following are two examples of how to use this command in user mode. Note that kernel mode usage is also possible but follows a different syntax.

As a simple example, assume that you have a structure whose type name is **MYTYPE**, and which has links within its **.links.Flink** and **.links.Blink** fields. You have a linked list that begins with the structure at 0x6BC000. The following extension command will go through the list and for each element will execute a **dd** L2 command. Because no address is being specified to the **dd** command, it will take the address of the list head as the desired address. This causes the first two DWORDS in each structure to be displayed.

```
dbgcmd
```

```
0:000> !list -t MYTYPE.links.Flink -x "dd" -a "L2" 0x6bc00
```

As a more complex example, consider the case of using **\$extret**. It follows the list of type **_LIST_ENTRY** at **RtlCriticalSectionList**. For each element, it displays the first four DWORDS, and then displays the **_RTL_CRITICAL_SECTION_DEBUG** structure located at an offset of eight bytes prior to the **Flink** element of the list entry.

```
dbgcmd
```

```
0:000> !list "-t ntdll!_LIST_ENTRY.Flink -e -x \"dd @$extret 14; dt
ntdll!_RTL_CRITICAL_SECTION_DEBUG @$extret-0x8\""
ntdll!RtlCriticalSectionList"
dd @$extret 14; dt ntdll!_RTL_CRITICAL_SECTION_DEBUG @$extret-0x8
7c97c0c8 7c97c428 7c97c868 01010000 00000080
+0x000 Type : 1
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : (null)
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x7c97c428 - 0x7c97c868 ]
+0x010 EntryCount : 0x1010000
+0x014 ContentionCount : 0x80
+0x018 Spare : [2] 0x7c97c100

dd @$extret 14; dt ntdll!_RTL_CRITICAL_SECTION_DEBUG @$extret-0x8
7c97c428 7c97c448 7c97c0c8 00000000 00000000
+0x000 Type : 0
+0x002 CreatorBackTraceIndex : 0
+0x004 CriticalSection : 0x7c97c0a0
+0x008 ProcessLocksList : _LIST_ENTRY [ 0x7c97c448 - 0x7c97c0c8 ]
+0x010 EntryCount : 0
+0x014 ContentionCount : 0
+0x018 Spare : [2] 0
```

!lmi

Article • 04/03/2024

The **!lmi** extension displays detailed information about a module.

dbgcmd
!lmi Module

Parameters

Module

Specifies a loaded module, either by name or by base address.

DLL

Dbghelp.dll

Remarks

Module addresses can be determined by using the [!Im \(List Loaded Modules\)](#) command.

The **!lmi** extension analyzes the module headers and displays a formatted summary of the information therein. If the module headers are paged out, an error message is displayed. To see a more extensive display of header information, use the [!dh](#) extension command.

This command shows a number of fields, each with a different title. Some of these titles have specific meanings:

- The **Image Name** field shows the name of the executable file, including the extension. Typically, the full path is included in user mode but not in kernel mode.
- The **Module** field shows the *module name*. This is usually just the file name without the extension. In a few cases, the module name differs significantly from the file name.
- The **Symbol Type** field shows information about the debugger's attempts to load this module's symbols. For an explanation of the various status values, see [Symbol](#)

Status Abbreviations. If symbols have been loaded, the symbol file name follows this.

- The first address in the module is shown as **Base Address**. The size of the module is shown as **Size**. Thus, if **Base Address** is "faab4000" and **Size** is "2000", the module extends from 0xFAAB4000 to 0xFAAB5FFF, inclusive.

Here is an example:

```
dbgcmd

0:000> lm
start   end     module name
00400000 0042d000  Prymes      C (pdb symbols)          Prymes.pdb
77e80000 77f35000  KERNEL32    (export symbols)
C:\WINNT\system32\KERNEL32.dll
77f80000 77ffb000  ntdll       (export symbols)          ntdll.dll

0:000> !lmi 00400000
Loaded Module Info: [00400000]
    Module: Prymes
    Base Address: 00400000
    Image Name: Prymes.exe
    Machine Type: 332 (I386)
    Time Stamp: 3c76c346 Fri Feb 22 14:16:38 2002
    Size: 2d000
    CheckSum: 0
    Characteristics: 230e stripped
    Debug Data Dirs: Type Size     VA  Pointer
                      MISC 110,     0,   77a00 [Data not mapped]
    Symbol Type: EXPORT - PDB not found
    Load Report: export symbols
```

For an explanation of the abbreviations shown on the **Characteristics** line of this example, see [Symbol Status Abbreviations](#).

Feedback

Was this page helpful?

 Yes

 No

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

!mui

Article • 10/25/2023

The **!mui** extension displays the Multilingual User Interface (MUI) cache information.

```
dbgcmd  
!mui -c  
!mui -s  
!mui -r ModuleAddress  
!mui -i  
!mui -f  
!mui -t  
!mui -u  
!mui -d ModuleAddress  
!mui -e ModuleAddress  
!mui -?
```

Parameters

-c

Causes the output to include the language identifier (ID), a pointer to the module, a pointer to the resource configuration data, and a pointer to the associated MUI DLL for each module.

-s

(Kernel Mode Only) Causes the display to include the full file paths for the module and associated MUI DLL for each module.

-r ** *ModuleAddress***

Causes the resource configuration data for the module at *ModuleAddress* to be displayed. This includes the file type, the checksum value, and the resource types.

-i

Causes the output to include the installed and licensed MUI languages and their associated information.

-f

Causes the output to include the loader merged language fallback list.

-t

Causes the output to include the thread preference language.

-u

Causes the output to include the current thread user UI language setting.

-d ** *ModuleAddress***

Causes the output to include contained resources for the module at *ModuleAddress*.

-e ** *ModuleAddress***

Causes the output to include contained resource types for the module at *ModuleAddress*.

-?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Windows XP	Unavailable
Windows Vista and later	Exts.dll

Additional Information

For information about MUI and resource configuration data format, see the Microsoft Windows SDK documentation.

!net_send

Article • 03/26/2024

The !net_send extension command is obsolete and is no longer being supported.

Feedback

Was this page helpful?

 Yes

 No

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

!obja

Article • 04/03/2024

The **!obja** extension displays the attributes of an object in the object manager.

dbgcmd

!obja Address

Parameters

Address

Specifies the hexadecimal address of the object header you want to examine.

DLL

Ext.dll

Additional Information

For information about objects and the object manager, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The attributes pertaining to the specified object are listed. Valid attributes are:

C++

```
#define OBJ_INHERIT          0x00000002L
#define OBJ_PERMANENT         0x00000010L
#define OBJ_EXCLUSIVE          0x00000020L
#define OBJ_CASE_INSENSITIVE   0x00000040L
#define OBJ_OPENIF              0x00000080L
#define OBJ_OPENLINK             0x00000100L
#define OBJ_VALID_ATTRIBUTES    0x000001F2L
```

Here is an example:

dbgcmd

```
kd> !obja 80967768
ObjA +80967768 at 80967768:
    OBJ_INHERIT
    OBJ_PERMANENT
    OBJ_EXCLUSIVE
```

Feedback

Was this page helpful?

 Yes

 No

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

!owner

Article • 04/03/2024

The **!owner** extension displays the owner of a module or function.

dbgcmd

```
!owner [Module[!Symbol]]
```

Parameters

Module

Specifies the module whose owner is desired. An asterisk (*) at the end of *Module* represents any number of additional characters.

Symbol

Specifies the symbol within *Module* whose owner is desired. An asterisk (*) at the end of *Symbol* represents any number of additional characters. If *Symbol* is omitted, the owner of the entire module is displayed.

DLL

Ext.dll

Remarks

If no parameters are used and a fault has occurred, **!owner** will display the name of the owner of the faulting module or function.

When you pass a module or function name to the **!owner** extension, the debugger displays the word **Followup** followed by the name of owner of the specified module or function.

For this extension to display useful information, you must first create a triage.ini file containing the names of the module and function owners.

For details on the triage.ini file and an example of the **!owner** extension, see [Specifying Module and Function Owners](#).

Feedback

Was this page helpful?

 Yes

 No

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

!peb

Article • 04/03/2024

The **!peb** extension displays a formatted view of the information in the process environment block (PEB).

```
dbgcmd
```

```
!peb [PEB-Address]
```

Parameters

PEB-Address

The hexadecimal address of the process whose PEB you want to examine. (This is not the address of the PEB as derived from the kernel process block for the process.) If *PEB-Address* is omitted in user mode, the PEB for the current process is used. If it is omitted in kernel mode, the PEB corresponding to the current [process context](#) is displayed.

DLL

Exts.dll

Additional Information

For information about process environment blocks, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The PEB is the user-mode portion of Microsoft Windows process control structures.

If the **!peb** extension with no argument gives you an error in kernel mode, you should use the [!process](#) extension to determine the PEB address for the desired process. Make sure your [process context](#) is set to the desired process, and then use the PEB address as the argument for **!peb**.

The exact output displayed depends on the Windows version and on whether you are debugging in kernel mode or user mode. The following example is taken from a kernel debugger attached to a Windows Server 2003 target:

```
dbgcmd
```

```
kd> !peb
PEB at 7ffdf000
  InheritedAddressSpace:      No
  ReadImageFileExecOptions:   No
  BeingDebugged:             No
  ImageBaseAddress:          4ad00000
  Ldr                      77fbe900
  Ldr.Initialized:           Yes
  Ldr.InInitializationOrderModuleList: 00241ef8 . 00242360
  Ldr.InLoadOrderModuleList:    00241e90 . 00242350
  Ldr.InMemoryOrderModuleList: 00241e98 . 00242358
      Base TimeStamp                 Module
  4ad00000 3d34633c Jul 16 11:17:32 2002 D:\WINDOWS\system32\cmd.exe
  77f40000 3d346214 Jul 16 11:12:36 2002 D:\WINDOWS\system32\ntdll.dll
  77e50000 3d3484ef Jul 16 13:41:19 2002
D:\WINDOWS\system32\kernel32.dll
.....
  SubSystemData:      00000000
  ProcessHeap:        00140000
  ProcessParameters: 00020000
  WindowTitle: "'D:\Documents and
Settings\Administrator\Desktop\Debuggers.lnk'"
  ImageFile:      'D:\WINDOWS\system32\cmd.exe'
  CommandLine:    '"D:\WINDOWS\system32\cmd.exe" '
  DllPath:         'D:\WINDOWS\system32;D:\WINDOWS\system32;.....
  Environment:    00010000
      ALLUSERSPROFILE=D:\Documents and Settings\All Users
      APPDATA=D:\Documents and Settings\UserTwo\Application Data
      CLIENTNAME=Console
.....
  windir=D:\WINDOWS
```

The similar **!teb** extension displays the thread environment block.

Feedback

Was this page helpful?

 Yes

 No

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

!rebase

Article • 04/03/2024

The **!rebase** extension searches in a rebase.log file for a specified address or symbol.

dbgcmd

```
!rebase [-r] Address [Path]
!rebase Symbol [Path]
!rebase -stack [Path]
!rebase -?
```

Parameters

-r

Attempts to load any module found in rebase.log.

Address

Specifies an address in standard hexadecimal format. The extension will search for DLLs near this address.

Path

Specifies the file path to the rebase.log. If *Path* is not specified, then the extension tries to guess the path to the rebase.log or, failing that, tries to read a rebase.log file from the current working directory.

Symbol

Specifies the symbol or image name. The extension will search for DLLs that contain this substring.

-stack

Displays all modules in the current stack.

-?

Displays a brief help text for this extension in the Debugger Command window.

DLL

Ext.dll

Feedback

Was this page helpful?

 Yes

 No

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

!rtlavl

Article • 04/03/2024

The **!rtlavl** extension displays the entries of an RTL_AVL_TABLE structure.

dbgcmd

```
!rtlavl Address [Module!Type]  
!rtlavl -?
```

Parameters

Address

Specifies the address of the RTL_AVL_TABLE to display.

Module

Specifies the module in which the data structure is defined.

Type

Specifies the name of a data structure.

-?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Ext.dll

Additional Information

Use the [!gentable](#) extension to display AVL tables.

Remarks

Including the *Module!Type* option causes each entry in the table to be interpreted as having the given type.

The display can be interrupted at any time by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD or CDB).

Feedback

Was this page helpful?

 Yes

 No

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

!sd

Article • 10/25/2023

The **!sd** extension displays the security descriptor at the specified address.

Syntax

```
dbgcmd
```

```
!sd Address [Flags]
```

Parameters

Address

Specifies the hexadecimal address of the SECURITY_DESCRIPTOR structure.

Flags

If this is set to 1, the friendly name is displayed. This includes the security identifier (SID) type, as well as the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For an application and an example of this command, see [Determining the ACL of an Object](#). For information about security descriptors, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see [!sid](#) and [!acl](#).

Remarks

Here is an example:

```
dbgcmd
```

```
kd> !sd e1a96a80 1
->Revision: 0x1
->Sbz1 : 0x0
->Control : 0x8004
```

```
    SE_DACL_PRESENT
    SE_SELF_RELATIVE
->Owner    : S-1-5-21-518066528-515770016-299552555-2981724 (User:
MYDOMAIN\myuser)
->Group    : S-1-5-21-518066528-515770016-299552555-513 (Group:
MYDOMAIN\Domain Users)
->Dacl     :
->Dacl     : ->AclRevision: 0x2
->Dacl     : ->Sbz1      : 0x0
->Dacl     : ->AclSize    : 0x40
->Dacl     : ->AceCount   : 0x2
->Dacl     : ->Sbz2      : 0x0
->Dacl     : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[0]: ->AceFlags: 0x0
->Dacl     : ->Ace[0]: ->AceSize: 0x24
->Dacl     : ->Ace[0]: ->Mask : 0x001f0003
->Dacl     : ->Ace[0]: ->SID: S-1-5-21-518066528-515770016-299552555-2981724
(User: MYDOMAIN\myuser)

->Dacl     : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[1]: ->AceFlags: 0x0
->Dacl     : ->Ace[1]: ->AceSize: 0x14
->Dacl     : ->Ace[1]: ->Mask : 0x001f0003
->Dacl     : ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT
AUTHORITY\SYSTEM)

->Sacl     : is NULL
```

!sid

Article • 10/25/2023

The **!sid** extension displays the security identifier (SID) at the specified address.

Syntax

```
dbgcmd
```

```
!sid Address [Flags]
```

Parameters

Address

Specifies the address of the SID structure.

Flags

If this is set to 1, the SID type, domain, and user name for the SID is displayed.

If this is set to 1, the friendly name is displayed. This includes the SID type, as well as the domain and user name for the SID.

DLL

Exts.dll

Additional Information

For information about SIDs, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, or *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see [!sd](#) and [!acl](#).

Remarks

Here are two examples, one without the friendly name shown, and one with:

```
dbgcmd
```

```
kd> !sid 0xe1bf35b8
SID is: S-1-5-21-518066528-515770016-299552555-513
```

```
kd> !sid 0xe1bf35b8 1
SID is: S-1-5-21-518066528-515770016-299552555-513 (Group: MYGROUP\Domain
Users)
```

!slist

Article • 04/03/2024

The **!slist** extension displays a singly-linked list (SList).

```
dbgcmd
```

```
!slist Address [ Symbol [Offset] ]  
!slist -?
```

Parameters

Address

Specifies the address of the SLIST_HEADER.

Symbol

Specifies the data type to use for display. If *Symbol* is specified, the debugger will assume that each node of the SList is an instance of this data type when displaying it.

Offset

Specifies the byte offset of the SList pointer within the structure.

-?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Exts.dll

Remarks

If you know the nature of the linked structures, the *Symbol* and *Offset* parameters are very useful. To see the difference, here are two examples; the first omits the *Symbol* and *Offset* parameters, while the second includes them.

```
dbgcmd
```

```
0:000> !slist ListHead  
SLIST HEADER:  
+0x000 Alignment      : a000a002643e8  
+0x000 Next            : 2643e8  
+0x004 Depth           : a
```

```
+0x006 Sequence : a
```

SLIST CONTENTS:

```
002643e8 002642c0 0000000a 6e676953 72757461
002642c0 00264198 00000009 6e676953 72757461
00264198 00264070 00000008 6e676953 72757461
00264070 00263f48 00000007 6e676953 72757461
00263f48 00261420 00000006 6e676953 72757461
00261420 002612f8 00000005 6e676953 72757461
002612f8 002611d0 00000004 6e676953 72757461
002611d0 002610a8 00000003 6e676953 72757461
002610a8 00260f80 00000002 6e676953 72757461
00260f80 00000000 00000001 6e676953 72757461
```

dbgcmd

```
0:000> !slist ListHead _PROGRAM_ITEM 0
```

SLIST HEADER:

```
+0x000 Alignment : a000a002643e8
+0x000 Next : 2643e8
+0x004 Depth : a
+0x006 Sequence : a
```

SLIST CONTENTS:

002643e8

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 0xa
+0x008 Description : [260] "Signature is: 10"
```

002642c0

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 9
+0x008 Description : [260] "Signature is: 9"
```

00264198

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 8
+0x008 Description : [260] "Signature is: 8"
```

00264070

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 7
+0x008 Description : [260] "Signature is: 7"
```

00263f48

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 6
+0x008 Description : [260] "Signature is: 6"
```

00261420

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 5
+0x008 Description : [260] "Signature is: 5"
```

002612f8

```
+0x000 ItemEntry : _SINGLE_LIST_ENTRY
+0x004 Signature : 4
+0x008 Description : [260] "Signature is: 4"
```

002611d0

```
+0x000 ItemEntry      : _SINGLE_LIST_ENTRY
+0x004 Signature       : 3
+0x008 Description     : [260]  "Signature is: 3"
002610a8
+0x000 ItemEntry      : _SINGLE_LIST_ENTRY
+0x004 Signature       : 2
+0x008 Description     : [260]  "Signature is: 2"
00260f80
+0x000 ItemEntry      : _SINGLE_LIST_ENTRY
+0x004 Signature       : 1
+0x008 Description     : [260]  "Signature is: 1"
```

Feedback

Was this page helpful?

 Yes

 No

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

!std_map

Article • 04/01/2024

The !std_map extension command is obsolete and is no longer being supported.

Feedback

Was this page helpful?

 Yes

 No

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

!stl

Article • 10/22/2024

The !stl extension command is obsolete and is no longer be supported. Use the debugger data model,

[dx \(Display Debugger Object Model Expression\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!str

Article • 04/01/2024

The **!str** extension is obsolete and is no longer supported. Use the debugger data model,

[dx \(Display Debugger Object Model Expression\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!sym

Article • 04/03/2024

The **!sym** extension controls noisy symbol loading and symbol prompts.

```
dbgcmd  
  
!sym  
!sym noisy  
!sym quiet  
!sym prompts  
!sym prompts off
```

Parameters

noisy

Activates noisy symbol loading.

quiet

Deactivates noisy symbol loading.

prompts

Allows authentication dialog boxes to appear when SymSrv receives an authentication request.

prompts off

Suppresses all authentication dialog boxes when SymSrv receives an authentication request. This may result in SymSrv being unable to access symbols over the internet.

DLL

Dbghelp.dll

Remarks

If the **!sym** extension is used with no arguments, the current state of noisy symbol loading and symbol prompting is displayed.

The **!sym noisy** and **!sym quiet** extensions control noisy symbol loading. For details and for other methods of displaying and changing this option, see [SYMOPT_DEBUG](#).

The **!sym prompts** and **!sym prompts off** extensions control whether authentication dialogs are displayed when SymSrv encounters an authentication request. These commands must be followed by [.reload \(Reload Module\)](#) for them to take effect. Authentication requests may be sent by proxy servers, internet firewalls, smart card readers, and secure websites. For details and for other methods of changing this option, see [Firewalls and Proxy Servers](#).

Note Noisy symbol loading should not be confused with noisy source loading -- that is controlled by the [.srcnoisy \(Noisy Source Loading\)](#) command.

Feedback

Was this page helpful?

 Yes

 No

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

!symsrv

Article • 04/01/2024

The !symsrv extension command is obsolete and is no longer being supported.

Feedback

Was this page helpful?

 Yes

 No

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

!teb

Article • 10/25/2023

The **!teb** extension displays a formatted view of the information in the thread environment block (TEB).

```
dbgcmd
```

```
!teb [TEB-Address]
```

Parameters

TEB-Address

The hexadecimal address of the thread whose TEB you want to examine. (This is not the address of the TEB as derived from the kernel thread block for the thread.) If *TEB-Address* is omitted in user mode, the TEB for the current thread is used. If it is omitted in kernel mode, the TEB corresponding to the current [register context](#) is displayed.

DLL

Exts.dll

Additional Information

For information about thread environment blocks, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The TEB is the user-mode portion of Microsoft Windows thread control structures.

If the **!teb** extension with no argument gives you an error in kernel mode, you should use the [!process](#) extension to determine the TEB address for the desired thread. Make sure your [register context](#) is set to the desired thread, and then use the TEB address as the argument for **!teb**.

Here is an example of this command's output in user mode:

```
dbgcmd
```

```
0:001> ~
 0  id: 324.458  Suspend: 1 Teb 7ffde000 Unfrozen
. 1  id: 324.48c  Suspend: 1 Teb 7ffdd000 Unfrozen

0:001> !teb
TEB at 7FFDD000
  ExceptionList:    76ffd0c
  Stack Base:      770000
  Stack Limit:     76f000
  SubSystemTib:    0
  FiberData:       1e00
  ArbitraryUser:   0
  Self:            7ffdd000
  EnvironmentPtr:  0
  ClientId:        324.48c
  Real ClientId:   324.48c
  RpcHandle:       0
  Tls Storage:     0
  PEB Address:     7ffdf000
  LastErrorValue:  0
  LastStatusValue: 0
  Count Owned Locks: 0
  HardErrorsMode:  0
```

The similar **!peb** extension displays the process environment block.

!tls

Article • 04/03/2024

The `!tls` extension displays a thread local storage (TLS) slot.

```
dbgcmd
```

```
!tls Slot [TEB]
```

Parameters

Slot

Specifies the TLS slot. This can be any value between 0 and 1088 (decimal). If *Slot* is -1, all slots are displayed.

TEB

Specifies the thread environment block (TEB). If this is 0 or omitted, the current thread is used.

DLL

Exts.dll

Remarks

Here is an example:

```
dbgcmd
```

```
0:000> !tls -1
TLS slots on thread: c08.f54
0x0000 : 00000000
0x0001 : 003967b8
0:000> !tls 0
c08.f54: 00000000
```

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!token

Article • 10/25/2023

The **!token** extension displays a formatted view of a security token object.

Kernel-Mode Syntax:

```
dbgcmd  
!token [-n] [Address]  
!token -?
```

User-Mode Syntax:

```
dbgcmd  
!token [-n] [Handle]  
!token -?
```

Parameters

Address

(Kernel mode only) Specifies the address of the token to be displayed. If this is 0 or omitted, the token for the active thread is displayed.

Handle

(User mode only) Specifies the handle of the token to be displayed. If this is 0 or omitted, the token associated with the target thread is displayed.

-n

(User mode only) Causes the display to include the friendly name. This includes the security identifier (SID) type, as well as the domain and user name for the SID. This option cannot be used when you are debugging LSASS.

-?

Displays help text for this extension.

DLL

Exts.dll

The **!token** command is available in kernel-mode and live user-mode debugging. It cannot be used on user-mode dump files.

Additional Information

For information about the kernel-mode TOKEN structure, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon. For information about the user-mode TOKEN structure, see the Microsoft Windows SDK documentation.

Remarks

The TOKEN structure is a security object type that represents an authenticated user process. Every process has an assigned token, which becomes the default token for each thread of that process. However, an individual thread can be assigned a token that overrides this default.

You can get the token address from the output of [!process](#). To display a list of the individual fields of the TOKEN structure, use the **dt nt!_TOKEN** command.

Here is an example:

```
dbgcmd

kd> !process 81464da8 1
PROCESS 81464da8 SessionId: 0 Cid: 03bc Peb: 7ffdf000 ParentCid: 0124
    DirBase: 0dec2000 ObjectTable: e1a31198 TableSize: 275.
    Image: MSMSGS.EXE
    VadRoot 81468cc0 Vads 170 Clone 0 Private 455. Modified 413. Locked 0.
    DeviceMap e1958438
    Token e1bed030
    ElapsedTime 0:44:15.0142
    UserTime 0:00:00.0290
    KernelTime 0:00:00.0300
    QuotaPoolUsage[PagedPool] 49552
    QuotaPoolUsage[NonPagedPool] 10872
    Working Set Sizes (now,min,max) (781, 50, 345) (3124KB, 200KB, 1380KB)
    PeakWorkingSetSize 1550
    VirtualSize 57 Mb
    PeakVirtualSize 57 Mb
    PageFaultCount 2481
    MemoryPriority BACKGROUND
    BasePriority 8
    CommitCharge 2497
kd> !exts.token -n e1bed030
_TOKEN e1bed030
TS Session ID: 0
User: S-1-5-21-518066528-515770016-299552555-2981724 (User: MYDOMAIN\myuser)
Groups:
```

00 S-1-5-21-518066528-515770016-299552555-513 (Group: MYDOMAIN\Domain Users)
 Attributes - Mandatory Default Enabled
01 S-1-1-0 (Well Known Group: localhost\Everyone)
 Attributes - Mandatory Default Enabled
02 S-1-5-32-544 (Alias: BUILTIN\Administrators)
 Attributes - Mandatory Default Enabled Owner
03 S-1-5-32-545 (Alias: BUILTIN\Users)
 Attributes - Mandatory Default Enabled
04 S-1-5-21-518066528-515770016-299552555-2999049 (Group: MYDOMAIN\AllUsers)
 Attributes - Mandatory Default Enabled
05 S-1-5-21-518066528-515770016-299552555-2931095 (Group: MYDOMAIN\SomeGroup1)
 Attributes - Mandatory Default Enabled
06 S-1-5-21-518066528-515770016-299552555-2931096 (Group: MYDOMAIN\SomeGroup2)
 Attributes - Mandatory Default Enabled
07 S-1-5-21-518066528-515770016-299552555-3014318 (Group: MYDOMAIN\SomeGroup3)
 Attributes - Mandatory Default Enabled
08 S-1-5-21-518066528-515770016-299552555-3053352 (Group: MYDOMAIN\Another Group)
 Attributes - Mandatory Default Enabled
09 S-1-5-21-518066528-515770016-299552555-2966661 (Group: MYDOMAIN\TestGroup)
 Attributes - Mandatory Default Enabled
10 S-1-5-21-2117033040-537160606-1609722162-17637 (Group: MYOTHERDOMAIN\someusers)
 Attributes - Mandatory Default Enabled
11 S-1-5-21-518066528-515770016-299552555-3018354 (Group: MYDOMAIN\TestGroup2)
 Attributes - Mandatory Default Enabled
12 S-1-5-21-518066528-515770016-299552555-3026602 (Group: MYDOMAIN\SomeGroup4)
 Attributes - Mandatory Default Enabled
13 S-1-5-21-518066528-515770016-299552555-2926570 (Group: MYDOMAIN\YetAnotherGroup)
 Attributes - Mandatory Default Enabled
14 S-1-5-21-661411660-2927047998-133698966-513 (Group: MYDOMAIN\Domain Users)
 Attributes - Mandatory Default Enabled
15 S-1-5-21-518066528-515770016-299552555-2986081 (Alias: MYDOMAIN\an_alias)
 Attributes - Mandatory Default Enabled GroupResource
16 S-1-5-21-518066528-515770016-299552555-3037986 (Alias: MYDOMAIN\AReallyLongGroupName1)
 Attributes - Mandatory Default Enabled GroupResource
17 S-1-5-21-518066528-515770016-299552555-3038991 (Alias: MYDOMAIN\AReallyLongGroupName2)
 Attributes - Mandatory Default Enabled GroupResource
18 S-1-5-21-518066528-515770016-299552555-3037999 (Alias: MYDOMAIN\AReallyLongGroupName3)
 Attributes - Mandatory Default Enabled GroupResource
19 S-1-5-21-518066528-515770016-299552555-3038983 (Alias:

```
MYDOMAIN\AReallyReallyLongGroupName)
    Attributes - Mandatory Default Enabled GroupResource
20 S-1-5-0-71188 (no name mapped)
    Attributes - Mandatory Default Enabled LogonId
21 S-1-2-0 (Well Known Group: localhost\LOCAL)
    Attributes - Mandatory Default Enabled
22 S-1-5-4 (Well Known Group: NT AUTHORITY\INTERACTIVE)
    Attributes - Mandatory Default Enabled
23 S-1-5-11 (Well Known Group: NT AUTHORITY\Authenticated Users)
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-518066528-515770016-299552555-513 (Group:
MYDOMAIN\Domain Users)
Privilges:
    00 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled
Default
    01 0x000000008 SeSecurityPrivilege              Attributes -
    02 0x000000011 SeBackupPrivilege                Attributes -
    03 0x000000012 SeRestorePrivilege               Attributes -
    04 0x00000000c SeSystemtimePrivilege            Attributes -
    05 0x000000013 SeShutdownPrivilege              Attributes -
    06 0x000000018 SeRemoteShutdownPrivilege        Attributes -
    07 0x000000009 SeTakeOwnershipPrivilege         Attributes -
    08 0x000000014 SeDebugPrivilege                 Attributes -
    09 0x000000016 SeSystemEnvironmentPrivilege     Attributes -
    10 0x00000000b SeSystemProfilePrivilege          Attributes -
    11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
    12 0x00000000e SeIncreaseBasePriorityPrivilege  Attributes -
    13 0x00000000a SeLoadDriverPrivilege             Attributes - Enabled
    14 0x00000000f SeCreatePagefilePrivilege          Attributes -
    15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
    16 0x000000019 SeUndockPrivilege                 Attributes - Enabled
    17 0x00000001c SeManageVolumePrivilege           Attributes -
Authentication ID:          (0,11691)
Impersonation Level:       Anonymous
TokenType:                  Primary
Source: User32              TokenFlags: 0x9 ( Token in use )
Token ID: 18296              ParentToken ID: 0
Modified ID:                (0, 18298)
RestrictedSidCount: 0       RestrictedSids: 00000000
```

!tp

Article • 04/03/2024

The **!tp** extension displays thread pool information.

```
dbgcmd
!tp pool Address [Flags]
!tp tqueue Address [Flags]
!tp ItemType Address [Flags]
!tp ThreadType [Address]
!tp stats Address [Flags]
!tp wfac Address
!tp wqueue Address Priority Node
!tp -?
```

Parameters

pool ** Address**

Causes the entire thread pool at *Address* to be displayed. If *Address* is 0, then all thread pools will be displayed.

tqueue ** Address**

Causes the active timer queue at *Address* to be displayed.

ItemType *Address*

Causes the specified thread pool item to be displayed. *Address* specifies the address of the item. *ItemType* specifies the type of the item; this can include any of the following possibilities:

obj

A generic pool item (such as an IO item) will be displayed.

timer

A timer item will be displayed.

wait

A wait item will be displayed.

work

A work item will be displayed.

ThreadType [*Address*]

Causes threads of the specified type to be displayed. If *Address* is included and nonzero,

then only the thread at this address is displayed. If *Address* is 0, all threads matching *ThreadType* are displayed. If *Address* is omitted, only the threads matching *ThreadType* associated with the current thread are displayed. *ThreadType* specifies the type of the thread to be displayed; this can include any of the following possibilities:

waiter

A thread pool waiter thread will be displayed.

worker

A thread pool worker thread will be displayed.

stats [Address]

Causes the debug statistics of the current thread to be displayed. *Address* may be omitted, but if it is specified, it must equal -1 (negative one), to represent the current thread.

wfac ** Address**

(Windows 7 and later only) Causes the worker factory at *Address* to be displayed. The specified *Address* must be a valid nonzero address.

wqueue ** Address**

(Windows 7 and later only) Causes display of a work queue and NUMA node that match the following: a specified priority, a specified NUMA node, and the pool, at a specified address, to which the NUMA node belongs. *Address* specifies the address of the pool. When the **wqueue** parameter is used, it must be followed by *Address*, *Priority*, and *Node*.

Priority

(Windows 7 and later only) Specifies the priority levels of the work queues to be displayed. *Priority* can be any of the following values:

0

Work queues with high priority are displayed.

1

Work queues with normal priority are displayed.

2

Work queues with low priority are displayed.

-1

All work queues are displayed.

Node

(Windows 7 and later only) Specifies a NUMA node belonging to the pool specified by

Address. If *Node* is -1 (negative one), all NUMA nodes are displayed.

Flags

Specifies what the display should contain. This can be a sum of any of the following bit values (the default is 0x0):

Bit 0 (0x1)

Causes the display to be single-line output. This bit value has no effect on the output when an *ItemType* is displayed.

Bit 1 (0x2)

Causes the display to include member information.

Bit 2 (0x4)

This flag is relevant only when the **pool** option is used. In Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008, this flag causes the display to include the pool work queue. In Windows 7 and later, this flag causes the display to include all the pool's work queues that are at normal priority, and all NUMA nodes.

-?

Displays a brief help text for this extension in the Debugger Command window.

DLL

Exts.dll

Additional Information

For information about thread pooling, see the Microsoft Windows SDK documentation.

Feedback

Was this page helpful?

 Yes

 No

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

!triage

Article • 04/01/2024

The **!triage** extension command is obsolete. Use [!analyze](#) instead.

Feedback

Was this page helpful?

 Yes

 No

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

!ustr

Article • 04/02/2024

The **!ustr** extension is obsolete and is no longer supported. Use the debugger data model,

[dx \(Display Debugger Object Model Expression\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!version

Article • 10/25/2023

The **!version** extension displays the version information for the extension DLL.

This extension command should not be confused with the [version \(Show Debugger Version\)](#) command.

```
dbgcmd
```

```
![ExtensionDLL.]version
```

Parameters

ExtensionDLL

Specifies the extension DLL whose version number is to be displayed.

DLL

This extension is available in most extension DLLs.

Remarks

If the extension DLL version does not match the debugger version, error messages will be displayed.

This extension command will not work on Windows XP and later versions of Windows. To display version information, use the [version \(Show Debugger Version\)](#) command.

The original purpose of this extension was to ensure that the DLL version matched the target version, since a mismatch would result in inaccurate results for many extensions. Newer DLLs are no longer restricted to working with only one version of Windows, so this extension is obsolete.

!winrterr

Article • 10/25/2023

The **!winrterr** sets the debugger reporting mode for Windows Runtime errors.

```
dbgcmd
```

```
!winrterr Mode  
!winrterr
```

Parameters

Mode

The following table describes the possible values for *Mode*.

Value	Description
report	When a Windows Runtime error occurs, the error and related text are displayed in the debugger, but execution continues. This is the default mode.
break	When a Windows Runtime error occurs, the error and related text are displayed in the debugger, and execution stops.
quiet	When a Windows Runtime error occurs, nothing is displayed in the debugger, and execution continues.

If *Mode* is omitted, **!winrterr** displays the current reporting mode. If the debugger has broken in as a result of a Windows Runtime error, the error and related text are also displayed.

See also

[Windows Runtime Debugger Commands](#)

[!hstring](#)

[!hstring2](#)

Kernel-Mode Extensions

Article • 10/25/2023

This section of the reference describes extension commands that are primarily used during kernel-mode debugging.

The debugger will automatically load the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions being used. See [Using Debugger Extension Commands](#) for a description of the default module search order. See [Loading Debugger Extension DLLs](#) for an explanation of how to load extension modules.

Each extension command reference page lists the DLLs that expose that command. Use the following rules to determine the proper directory from which to load this extension DLL:

- If your target computer is running Windows XP or a later version of Windows, use `winxp\Kdexts.dll`.

In addition, kernel-mode extensions that are not specific to any single operating system can be found in `winext\kext.dll`.

!ahcache

Article • 01/31/2024

The **!ahcache** extension displays the application compatibility cache.

```
dbgcmd
```

```
!ahcache [Flags]
```

Parameters

Flags

Specifies the information to include in the display. This can be any combination of the following bits (the default is zero):

Bit 0 (0x1)

Displays the RTL_GENERIC_TABLE list instead of the LRU list.

Bit 4 (0x10)

Verbose display: includes all entry details, not just the names.

DLL

Kdexts.dll



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

!alignmentfaults

Article • 01/31/2024

The `!alignmentfaults` extension displays all current type alignment faults by location and image, sorted by frequencies.

```
dbgcmd
```

```
!alignmentfaults
```

DLL

Kdexts.dll

Additional Information

For information about alignment faults, see the Microsoft Windows SDK documentation.

Remarks

This is only available on older versions of Windows before Windows 10 version 1803, that provided checked builds.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!analyzebugcheck

Article • 10/25/2023

The **!analyzebugcheck** extension command is obsolete. Use [!analyze](#) instead.

!apc

Article • 04/03/2024

The **!apc** extension formats and displays the contents of one or more asynchronous procedure calls (APCs).

dbgcmd

```
!apc
!apc proc Process
!apc thre Thread
!apc KAPC
```

Parameters

Process

Specifies the address of the process whose APCs are to be displayed.

Thread

Specifies the address of the thread whose APCs are to be displayed.

KAPC

Specifies the address of the kernel APC to be displayed.

DLL

Kdexts.dll

Additional Information

For information about APCs, see the Windows Driver Kit (WDK) documentation and Microsoft Windows Internals by Mark Russinovich and David Solomon.

Remarks

Without any parameters, **!apc** displays all APCs.

Here is an example:

Console

```
kd> !apc
*** Enumerating APCs in all processes
Process e0000000858ba8b0 System
Process e0000165fff86040 smss.exe
Process e0000165fff8c040 csrss.exe
Process e0000165fff4e1d0 winlogon.exe
Process e0000165fff101d0 services.exe
Process e0000165ffffa81d0 lsass.exe
Process e0000165fff201d0 svchost.exe
Process e0000165fff8e040 svchost.exe
Process e0000165fff3e040 svchost.exe
Process e0000165fff6e040 svchost.exe
Process e0000165fff24040 spoolsv.exe
Process e000000085666640 wmicl.exe
Process e00000008501e520 wmicl.exe
Process e0000000856db480 explorer.exe
Process e0000165fff206a0 ctfmon.exe
Process e0000000850009d0 ctfmon.exe
Process e0000165fff51600 conime.exe
Process e000000085496340 taskmgr.exe
Process e000000085489c30 userinit.exe
```

Feedback

Was this page helpful?

 Yes

 No

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

!apicerr

Article • 04/03/2024

The **!apicerr** extension displays the local Advanced Programmable Interrupt Controller (APIC) error log.

```
dbgcmd
```

```
!apicerr [Format]
```

Parameters

Format

Specifies the order in which to display the error log contents. This can be any one of the following values:

0x0

Displays the error log according to order of occurrence.

0x1

Displays the error log according to processor.

DLL

Kdexts.dll

This extension command can only be used with an x86-based or an x64-based target computer.

Additional Information

For information about APICs, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Feedback

Was this page helpful?

 Yes

 No

!arbinst

Article • 04/03/2024

The **!arbinst** extension displays information about a specified arbiter.

dbgcmd

```
!arbinst Address [Flags]
```

Parameters

Address

Specifies the hexadecimal address of the arbiter to be displayed.

Flags

Specifies how much information to display for each arbiter. At present, the only flag is 0x100. If this flag is set, then the aliases are displayed.

DLL

Kdexts.dll

Additional Information

See also the [!arbiter](#) extension.

Remarks

For the arbiter specified, **!arbinst** displays each allocated range of system resources, some optional flags, the PDO attached to that range (in other words, the range's owner), and the service name of this owner (if known).

Here is an example:

Console

```
kd> !arbinst e0000106002ee8e8
Port Arbiter "PCI I/O Port (b=02)" at e0000106002ee8e8
Allocated ranges:
 0000000000000000 - 0000000000001fff      00000000 <Not on bus>
 0000000000002000 - 00000000000020ff      P e0000000858bea20 (ql1280)
```

```
0000000000003000 - ffffffffffffffff 00000000 <Not on bus>
Possible allocation:
< none >
kd> !arbinst e0000106002ec458
Memory Arbiter "PCI Memory (b=02)" at e0000106002ec458
Allocated ranges:
0000000000000000 - 00000000ebfffff 00000000 <Not on bus>
0000000effdef00 - 00000000effdefff B e0000000858be560
0000000effdf000 - 00000000effdffff e0000000858bea20 (ql1280)
00000000f0000000 - ffffffffffffffff 00000000 <Not on bus>
Possible allocation:
< none >
```

Feedback

Was this page helpful?

 Yes

 No

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

!arbiter

Article • 10/25/2023

The **!arbiter** extension displays the current system resource arbiters and arbitrated ranges.

```
dbgcmd
```

```
!arbiter [Flags]
```

Parameters

Flags

Specifies which classes of arbiters are displayed. If omitted, all arbiters are displayed. These bits can be combined freely.

Bit 0 (0x1)

Display I/O arbiters.

Bit 1 (0x2)

Display memory arbiters.

Bit 2 (0x4)

Display IRQ arbiters.

Bit 3 (0x8)

Display DMA arbiters.

Bit 4 (0x10)

Display bus number arbiters.

Bit 8 (0x100)

Do not display aliases.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command.

Remarks

For each arbiter, **!arbiter** displays each allocated range of system resources, some optional flags, the PDO attached to that range (in other words, the range's owner), and the service name of this owner (if known).

The flags have the following meanings:

Flag	Meaning
S	Range is shared
C	Range in conflict
B	Range is boot-allocated
D	Range is driver-exclusive
A	Range alias
P	Range positive decode

Here is an example:

```
Console

kd> !arbiter 4

DEVNODE 80e203b8 (HTREE\ROOT\0)
  Interrupt Arbiter "" at 80167140
    Allocated ranges:
      0000000000000000 - 0000000000000000   B   80e1d3d8
      0000000000000001 - 0000000000000001   B   80e1d3d8
      ....
      000000000001a2 - 000000000001a2
      000000000001a2 - 0000000000001a2   CB   80e1d3d8
      00000000000001a2 - 00000000000001a2   CB   80e52538 (Serial)
      00000000000001a3 - 00000000000001a3       80e52778 (i8042prt)
      00000000000001b3 - 00000000000001b3       80e1b618 (i8042prt)
    Possible allocation:
      < none >
```

In this example, the next-to-last line shows the resource range (which consists of 0x1A3 alone), the PDO of 0x80E52778, and the service of i8042prt.sys. No flags are listed on this line.

You can now use **!devobj** with this PDO address to find the device extension and device node addresses:

Console

```
kd> !devobj 80e52778
Device object (80e52778) is for:
00000034 \Driver\PnpManager DriverObject 80e20610
Current Irp 00000000 RefCount 1 Type 00000004 Flags 00001040
DevExt 80e52830 DevObjExt 80e52838 DevNode 80e52628
ExtensionFlags (0000000000)
AttachedDevice (Upper) 80d78b28 \Driver\i8042prt
Device queue is not busy.
```

!ate

Article • 01/31/2024

The **!ate** extension displays the alternate page table entry (ATE) for the specified address.

```
dbgcmd
```

```
!ate Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address

Specifies the ATE to display.

DLL

Windows XP and later - Kdexts.dll

Additional Information

For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension is only available on Itanium-based computers.

The status flags for the ATE are shown in the following table. The **!ate** display indicates these bits with capital letters or dashes and adds additional information as well.

 Expand table

Display when set	Display when clear	Meaning
V	-	Commit.

Display when set	Display when clear	Meaning
G	-	Not accessed.
E	-	Execute.
W	R	Writable or read-only.
L	-	Locked. The ATE is locked, therefore, any faults on the page that contain the ATE will be retried until the current fault is satisfied. This can happen on multi-processor systems.
Z	-	Fill zero.
N	-	No access.
C	-	Copy on Write.
I	-	PTE indirect. This ATE indirectly references another physical page. The page that contains the ATE might have two conflicting ATE attributes.
P		Reserved.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!bcb

Article • 10/25/2023

The **!bcb** extension displays the specified buffer control block.

dbgcmd
!bcb Address

Parameters

Address

Specifies the address of the buffer control block.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see Remarks section)

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, use the [!cchelp](#) extension.

Remarks

This extension is available for Windows 2000 only. In Windows XP or later, use the [dt nt!_BCB](#) command to display the buffer control block directly.

!blockeddrv

Article • 04/03/2024

The `!blockeddrv` extension displays the list of blocked drivers on the target computer.

```
dbgcmd
```

```
!blockeddrv
```

DLL

Kdexts.dll

Remarks

Here is an example:

```
dbgcmd
```

```
kd> !blockeddrv
Driver:      Status   GUID
afd.sys       0:      {00000008-0206-0001-0000-000030C964E1}
agp440.sys    0:      {0000005C-175A-E12D-5000-010020885580}
atapi.sys     0:      {0000005C-B04A-E12E-5600-000020885580}
audstub.sys   0:      {0000005C-B04A-E12E-5600-000020885580}
Beep.SYS      0:      {0000005C-B04A-E12E-5600-000020885580}
Cdfs.SYS     0:      {00000008-0206-0001-0000-000008F036E1}
.....
```

Feedback

Was this page helpful?

 Yes

 No

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

!bpid

Article • 01/31/2024

The **!bpid** extension requests that a process on the target computer break into the debugger or requests that a user-mode debugger be attached to a process on the target computer.

dbgcmd

```
!bpid [Options] PID
```

Parameters

Option

Controls the additional activities of this command.

The valid values for *Option* appear in the following table.

[] [Expand table](#)

-a	Attaches a new user-mode debugger to the process specified by <i>PID</i> . The user-mode debugger runs on the target machine.
-s	Adds a breakpoint that occurs in the WinLogon process immediately before the break in the user-mode process specified by <i>PID</i> . This allows the user to verify the request before attempting the action.
-w	Stores the request in the memory in the target computer. The target system can then repeat the request, but this is not usually necessary.

PID

Specifies the process ID of the desired process on the target computer. If you are using this to control a user-mode debugger on the target computer, *PID* should be the process ID of the target application, not of the user-mode debugger. (Because process IDs are usually listed in decimal format, you might need to prefix this with **0n** or convert it to hexadecimal format.)

DLL

This extension command is supported on x86-based, x64-based target computers.

Remarks

This command is especially useful when redirecting input and output from a user-mode debugger to the kernel debugger. It causes the user-mode target application to break into the user-mode debugger, which in turn requests input from the kernel debugger. See [Controlling the User-Mode Debugger from the Kernel Debugger](#) for details.

If this command is used in another situation, the user-mode process calls **DbgBreakPoint**. This will usually break directly into the kernel debugger.

The **-s** option causes a break in WinLogon just before the break in the specified process occurs. This is useful if you want to perform debugging actions within WinLogon's process context. The [**g \(Go\)**](#) command can then be used to move on to the second break.

Note that there are ways in which this extension can fail to execute:

- Lack of resources. The **!bpid** extension injects a thread into the target process, so the system must have enough resources to create a thread. Using the **-a** option requires even more system resources since **!bpid -a** must run a full instance of a debugger on the target computer.
- The loader lock is already held. Both **!bpid** and **!bpid -a** require a thread to run in the target process in order to make it break into the debugger. If another thread is holding the loader lock, then the **!bpid** thread will not be able to run and a break into the debugger may not occur. Thus, if **!bpid** fails when there is enough user-mode memory available for the target process, it is possible that the loader lock is held.
- Lack of permission. The operation of the **!bpid** extension requires permission sufficient for WinLogon to create a remote thread and attach a debugger to a given process.
- No access to ntsd.exe. If ntsd.exe is not found in a commonly known path, **!bpid** will fail to set an appropriate PID. Note that ntsd.exe is not included by default with Windows Vista.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!btb

Article • 10/25/2023

The **!btb** extension displays the Itanium-based processor, branch traces buffer (BTB) configuration and trace registers for the current processor.

```
dbgcmd
```

```
!btb
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

DLL

Windows XP and later

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

!bth

Article • 04/03/2024

The **!bth** extension displays the Itanium-based branch traces history for the specified processor.

```
dbgcmd
```

```
!bth [Processor]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor

Specifies a processor. If *Processor* is omitted, then the branch trace history for all of processors is displayed.

DLL

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Feedback

Was this page helpful?

 Yes

 No

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

!bugdump

Article • 10/25/2023

The **!bugdump** extension formats and displays the information contained in the bug check callback buffers.

dbgsyntax

```
!bugdump [Component]
```

Parameters

Component

Specifies the component whose callback data is to be examined. If omitted, all bug check callback data is displayed.

DLL

Kdexts.dll

Additional Information

For more information, see [Reading Bug Check Callback Data](#).

Remarks

This extension can only be used after a bug check has occurred, or when you are debugging a kernel-mode crash dump file.

The *Component* parameter corresponds to the final parameter used in [KeRegisterBugCheckCallback](#).

The buffers that hold callback data are not available in a Small Memory Dump. These buffers are present in Kernel Memory Dumps and Full Memory Dumps. However, in Windows XP SP1, Windows Server 2003, and later versions of Windows, the dump file is created before the drivers' **BugCheckCallback** routines are called, and therefore these buffers will not contain the data written by these routines.

If you are performing live debugging of a crashed system, all callback data will be present.

!bushnd

Article • 10/25/2023

The **!bushnd** extension displays a HAL BUS_HANDLER structure.

dbgsyntax

```
!bushnd [Address]
```

Parameters

Address

Specifies the hexadecimal address of the HAL BUS_HANDLER structure. If omitted, **!bushnd** displays a list of buses and the base address of the handler.

DLL

Kdexts.dll

!ca

Article • 10/25/2023

The **!ca** extension displays information about a control area.

dbgsyntax

```
!ca [Address | 0 | -1] [Flags]
```

Parameters

Address

Address of the control area. If you specify 0 for this parameter, information is displayed about all control areas. If you specify -1 for this parameter, information is displayed about the unused segment list.

Flags

Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

Flag	Description
0x1	Display segment information.
0x2	Display subsection information.
0x4	Display the list of mapped views. (Windows 7 and later)
0x8	Display compact (single line) output.
0x10	Display file-backed control areas.
0x20	Display control areas backed by the page file.
0x40	Display image control areas.

If none of the last three flags are specified, all three types of control area are displayed.

DLL

Kdexts.dll

Additional Information

For information about control areas, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

To get a list of the control areas of all mapped files, use the [!memusage](#) extension.

Here is an example:

```
dbgcmd

kd> !memusage
loading PFN database
loading (99% complete)
    Zeroed:      16 (     64 kb)
        Free:       0 (      0 kb)
    Standby:   2642 ( 10568 kb)
    Modified:    720 ( 2880 kb)
ModifiedNoWrite:      0 (      0 kb)
Active/Valid: 13005 ( 52020 kb)
Transition:      0 (      0 kb)
Unknown:        0 (      0 kb)
    TOTAL: 16383 ( 65532 kb)

Building kernel map
Finished building kernel map

Usage Summary (in Kb):
Control  Valid  Standby  Dirty  Shared  Locked  PageTables  name
ff8636e8    56     376      0      0      0      0  mapped_file( browseui.dll )
ff8cf388    24      0      0      0      0      0  mapped_file( AVH32DLL.DLL )
ff8d62c8    12      0      0      0      0      0  mapped_file( PSAPI.DLL )
ff8dd468   156     28      0      0      0      0  mapped_file( INOJOBSV.EXE )
fe424808   136     88      0     52      0      0  mapped_file( oleaut32.dll )
fe4228a8   152     44      0    116      0      0  mapped_file( MSVCRT.DLL )
ff8ec848     4      0      0      0      0      0  No Name for File
ff859de8     0     32      0      0      0      0  mapped_file( timedate.cpl )
. . .

kd> !ca ff8636e8

ControlArea @ff8636e8
Segment: e1b74548  Flink                  0  Blink:                0
Section Ref      0  Pfn Ref                 6c  Mapped Views:      1
User Ref         1  Subsections              5  Flush Count:        0
File Object ff86df88  ModWriteCount        0  System Views:      0
WaitForDel       0  Paged Usage             380 NonPaged Usage:  e0
Flags (10000a0)  Image File HadUserReference

File: \WINNT\System32\browseui.dll
```

Segment @ e1b74548:

Base address	0	Total Ptes	c8	NonExtendPtes:	c8
Image commit	1	ControlArea	ff8636e8	SizeOfSegment:	c8000
Image Base	0	Committed	0	PTE Template:	31b8438
Based Addr	76e10000	ProtoPtes	e1b74580	Image Info:	e1b748a4

Subsection 1. @ ff863720

ControlArea:	ff8636e8	Starting Sector	0	Number Of Sectors	2
Base Pte	e1b74580	Ptes In subsect	1	Unused Ptes	0
Flags	15	Sector Offset	0	Protection	1
ReadOnly CopyOnWrite					

Subsection 2. @ ff863740

ControlArea:	ff8636e8	Starting Sector	2	Number Of Sectors	3d0
Base Pte	e1b74584	Ptes In subsect	7a	Unused Ptes	0
Flags	35	Sector Offset	0	Protection	3
ReadOnly CopyOnWrite					

Subsection 3. @ ff863760

ControlArea:	ff8636e8	Starting Sector	3D2	Number Of Sectors	7
Base Pte	e1b7476c	Ptes In subsect	1	Unused Ptes	0
Flags	55	Sector Offset	0	Protection	5
ReadOnly CopyOnWrite					

Subsection 4. @ ff863780

ControlArea:	ff8636e8	Starting Sector	3D9	Number Of Sectors	21f
Base Pte	e1b74770	Ptes In subsect	44	Unused Ptes	0
Flags	15	Sector Offset	0	Protection	1
ReadOnly CopyOnWrite					

Subsection 5. @ ff8637a0

ControlArea:	ff8636e8	Starting Sector	5F8	Number Of Sectors	3a
Base Pte	e1b74880	Ptes In subsect	8	Unused Ptes	0
Flags	15	Sector Offset	0	Protection	1
ReadOnly CopyOnWrite					

!callback

Article • 10/25/2023

The **!callback** extension displays the callback data related to the trap for the specified thread.

dbgsyntax

```
!callback Address [Number]
```

Parameters

Address

Specifies the hexadecimal address of the thread. If this is -1 or omitted, the current thread is used.

Number

Specifies the number of the desired callback frame. This frame is noted in the display.

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For information about system traps, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If the system has not experienced a system trap, this extension will not produce useful data.

!calldata

Article • 10/25/2023

The **!calldata** extension displays performance information in the form of procedure call statistics from the named table.

dbgsyntax

```
!calldata Table
```

Parameters

Table

Name of the table that collects the call data.

DLL

Kdexts.dll

!can_write_kdump

Article • 10/25/2023

The `!can_write_kdump` extension verifies that there is enough disk space on the target computer to write a kernel dump file of the specified type.

dbgsyntax

```
!can_write_kdump [-dn] [Options]
```

Parameters

-dn

Specifies that the file system on the target computer is an NTFS file system. If this parameter is omitted, then the amount of disk free space cannot be determined, and a warning will be shown. However, the amount of space required will still be displayed.

Options The following options are valid:

-t

Specifies that the extension should determine if there is enough space for a minidump.

-s

Specifies that the extension should determine if there is enough space for a summary kernel dump. This is the default value.

-f

Specifies that the extension should determine if there is enough space for a full kernel dump.

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

Remarks

If no *Option* is specified, then the extension will determine if there is enough space for a summary kernel dump.

In the following example, the file system is not specified:

```
dbgcmd

kd> !can_write_kdump
Checking kernel summary dump...
WARNING: Can't predict how many pages will be used, assuming worst-case.
Physical memory: 285560 KB
Page file size: 1572864 KB
NO: Page file too small
```

!cbreg

Article • 04/03/2024

The **!cbreg** extension displays CardBus Socket registers and CardBus Exchangable Card Architecture (ExCA) registers.

dbgsyntax

```
!cbreg [%]Address
```

Parameters

%%

Indicates that *Address* is a physical address rather than a virtual address.

Address

Specifies the address of the register to be displayed.

DLL

Kext.dll

The **!cbreg** extension is only available for an x86-based target computer.

Additional Information

The [!exca](#) extension can be used to display PCIC ExCA registers by socket number.

Feedback

Was this page helpful?

 Yes

 No

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

!cchelp

Article • 10/25/2023

The **!cchelp** extension displays some brief Help text in the Debugger command window for some of the cache management extensions.

```
dbgsyntax
```

```
!cchelp
```

DLL

Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

The **!cchelp** extension displays help for the [!bcb](#), [!defwrites](#), [!finddata](#), and [!scm](#) cache management extensions. Other cache management extensions include [!openmaps](#) and [!pcm](#).

!chklowmem

Article • 10/25/2023

The **!chklowmem** extension determines whether physical memory pages below 4 GB are filled with the required fill pattern on a computer that was booted with the [/pae](#) and [/nolowmem](#) options.

dbgsyntax

```
!chklowmem
```

DLL

Kdexts.dll

Remarks

This extension is useful when you are verifying that kernel-mode drivers operate properly with physical memory above the 4 GB boundary. Typically, drivers fail by truncating a physical address to 32 bits and then in writing below the 4 GB boundary. The **!chklowmem** extension will detect any writes below the 4 GB boundary.

!cmreslist

Article • 10/25/2023

The **!cmreslist** extension displays the CM_RESOURCE_LIST structure for the specified device object.

dbgsyntax

```
!cmreslist Address
```

Parameters

Address

Specifies the hexadecimal address of the CM_RESOURCE_LIST structure.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about the CM_RESOURCE_LIST structure, see the Windows Driver Kit (WDK) documentation.

!cpuinfo

Article • 10/25/2023

The **!cpuinfo** extension displays detailed information about the target computer's CPU.

Syntax

```
dbgsyntax
```

```
!cpuinfo [Processor]
```

Parameters

Processor

Specifies the processor to be displayed. If this is omitted, all processors are displayed.

DLL

Kdexts.dll

Additional Information

For more information about debugging multiprocessor computers, see [Multiprocessor Syntax](#).

Remarks

The **!cpuinfo** extension command can be used when performing [local kernel debugging](#).

Here is an example generated by an x86-based processor:

```
dbgcmd
```

```
kd> !cpuinfo
CP F/M/S Manufacturer MHz Update Signature Features
 0 6,1,9 GenuineIntel 198 000000d200000000 000000ff
```

The **CP** column indicates the processor number. The **Manufacturer** column specifies the processor manufacturer. The **MHz** or **Speed** column specifies the speed, in MHz, of the processor, if it is available.

For an x86-based processor or an x64-based processor, the **F** column displays the processor family number, the **M** column displays the processor model number, and the **S** column displays the stepping size.

Other columns will also appear, depending on your machine's specific architecture.

For details on how to interpret specific values for each entry, as well as any additional columns, consult the processor manual.

!db, !dc, !dd, !dp, !dq, !du, !dw

Article • 10/25/2023

The **!db**, **!dc**, **!dd**, **!dp**, **!dq**, **!du**, and **!dw** extensions display data at the specified physical address on the target computer.

These extension commands should not be confused with the [d* \(Display Memory\)](#) command, or with the [!ntsdexts.dp](#) extension command.

dbgcmd

```
!db [Caching] [-m] [PhysicalAddress] [L Size]
!dc [Caching] [-m] [PhysicalAddress] [L Size]
!dd [Caching] [-m] [PhysicalAddress] [L Size]
!dp [Caching] [-m] [PhysicalAddress] [L Size]
!dq [Caching] [-m] [PhysicalAddress] [L Size]
!du [Caching] [-m] [PhysicalAddress] [L Size]
!dw [Caching] [-m] [PhysicalAddress] [L Size]
```

Parameters

Caching

Can be any one of the following values. The *Caching* value must be surrounded by square brackets:

[c]

Causes this extension to read from cached memory.

[uc]

Causes this extension to read from uncached memory.

[wc]

Causes this extension to read from write-combined memory.

-m

Causes memory to be read one unit at a time. For example, **!db -m** reads memory in 8-bit chunks and **!dw -m** reads memory in 16-bit chunks. If your hardware does not support 32-bit physical memory reads, it may be necessary to use the **-m** option. This option does not affect the length or appearance of the display -- it only affects how the memory is accessed.

PhysicalAddress

Specifies the first physical address to be displayed, in hexadecimal format. If this is

omitted the first time this command is used, the address defaults to zero. If this is omitted on a subsequent use, the display will begin where the last display ended.

L **** Size

Specifies the number of chunks of memory to display. The size of a chunk is determined by the precise extension used.

Environment

Mode	Kernel Mode
------	-------------

DLL

Kext.dll

Additional Information

To write to physical memory, use the [!e*](#) extensions. For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Remarks

These extensions each display physical memory, but their display formats and default length differ:

- The **!db** extension displays hexadecimal bytes and their ASCII character equivalents. The default length is 128 bytes.
- The **!dc** extension displays DWORD values and their ASCII character equivalents. The default length is 32 DWORDs (128 total bytes).
- The **!dd** extension displays DWORD values. The default length is 32 DWORDs (128 total bytes).
- The **!dp** extension displays ULONG_PTR values. These are either 32-bit or 64-bit words, depending on the instruction size. The default length is 128 total bytes.
- The **!dq** extension displays ULONG64_PTR values. These are 32-bit words. The default length is 128 total bytes.

- The **!du** extension displays UNICODE characters. The default length is 16 characters (32 total bytes), or until a NULL character is encountered.
- The **!dw** extension displays WORD values. The default length is 64 DWORDs (128 total bytes).

Consequently, using two of these extensions that are distinct with the same value of *Size* will most likely result in a difference in the total amount of memory displayed. For example, using the command **!db L 32** results in 32 bytes being displayed (as hexadecimal bytes), whereas the command **!dd L 32** results in 128 bytes being displayed (as DWORD values).

Here is an example in which the caching attribute flag is needed:

```
dbgcmd
```

```
kd> !dc e9000
physical memory read at e9000 failed
If you know the caching attributes used for the memory,
try specifying [c], [uc] or [wc], as in !dd [c] <params>.
WARNING: Incorrect use of these flags will cause unpredictable
processor corruption. This may immediately (or at any time in
the future until reboot) result in a system hang, incorrect data
being displayed or other strange crashes and corruption.

kd> !dc [c] e9000
#   e9000 000ea002 000ea002 000ea002 000ea002 .....
#   e9010 000ea002 000ea002 000ea002 000ea002 .....
```

!dbgprint

Article • 10/25/2023

The **!dbgprint** extension displays a string that was previously sent to the **DbgPrint** buffer.

```
dbgcmd
```

```
!dbgprint
```

DLL

Kdexts.dll

Additional Information

For information about **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx**, see [Sending Output to the Debugger](#).

Remarks

The kernel-mode routines **DbgPrint**, **KdPrint**, **DbgPrintEx**, and **KdPrintEx** send a formatted string to a buffer on the target computer. The string is automatically displayed in the Debugger Command window on the host computer unless such printing has been disabled.

Generally, messages sent to this buffer are displayed automatically in the Debugger Command window. However, this display can be disabled through the Global Flags (gflags.exe) utility. Moreover, this display does not automatically appear during local kernel debugging. For more information, see "The DbgPrint Buffer" in [Reading and Filtering Debugging Messages](#).

The **!dbgprint** extension causes the contents of this buffer to be displayed (regardless of whether automatic printing has been disabled). It will not show messages that have been filtered out based on their component and importance level. (For details on this filtering, see [Reading and Filtering Debugging Messages](#).)

!dblink

Article • 10/25/2023

The **!dblink** extension displays a linked list in the backward direction.

dbgcmd

```
!dblink Address [Count] [Bias]
```

Parameters

Address

Specifies the address of a LIST_ENTRY structure. The display will begin with this node.

Count

Specifies the maximum number of list entries to display. If this is omitted, the default is 32.

Bias

Specifies a mask of bits to ignore in each pointer. Each **Blink** address is ANDed with (NOT *Bias*) before following it to the next location. The default is zero (in other words, do not ignore any bits).

DLL

Kdexts.dll

Remarks

The **!dblink** extension traverses the **Blink** fields of the LIST_ENTRY structure and displays up to four ULONGs at each address. To go in the other direction, use [!dflink](#).

The [dl \(Display Linked List\)](#) command is more versatile than **!dblink** and [!dflink](#).

!dcr

Article • 04/03/2024

The **!dcr** extension displays the default control register (DCR) at the specified address.

dbgcmd

```
!dcr Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Expression

Specifies the hexadecimal address of the DCR to display. The expression **@dcr** can also be used for this parameter. In that case, information about the current processor DCR is displayed.

DisplayLevel

Can be any one of the following options:

0

Causes only the values of each DCR field to be displayed. This is the default value.

1

Causes the display to include more in-depth information about each of the DCR fields that is not reserved or ignored.

2

Causes the display to include more in-depth information about all of the DCR fields, including those that are ignored or reserved.

DLL

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

The DCR specifies default parameters for the processor status register values on interruption. The DCR also specifies some additional global controls, as well as whether or not speculative load faults can be deferred.

Here are a couple of examples:

```
dbgcmd

kd> !dcr @dcr
dcr:pp be lc dm dp dk dx dr da dd
1 0 1 1 1 1 1 1 1 1

kd> !dcr @dcr 2

pp : 1 : Privileged Performance Monitor Default
be : 0 : Big-Endian Default
lc : 1 : IA-32 Lock check Enable
rv : 0 : reserved1
dm : 1 : Defer TLB Miss faults only
dp : 1 : Defer Page Not Present faults only
dk : 1 : Defer Key Miss faults only
dx : 1 : Defer Key Permission faults only
dr : 1 : Defer Access Rights faults only
da : 1 : Defer Access Bit faults only
dd : 0 : Defer Debug faults only
rv : 0 : reserved2
```

Feedback

Was this page helpful?

 Yes

 No

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

!dcs

Article • 10/25/2023

The **!dcs** extension is obsolete. To display the PCI configuration space, use [**!pci 100Bus Device Function**](#).

!deadlock

Article • 01/31/2024

The **!deadlock** extension displays information about deadlocks collected by the **Deadlock Detection** option of Driver Verifier.

```
dbgcmd
!deadlock
!deadlock 1
```

DLL

Windows XP and later - Kdexts.dll

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation.

Remarks

This extension will only provide useful information if Driver Verifier's **Deadlock Detection** option has detected a lock hierarchy violation and issued [bug check 0xC4](#) (DRIVER_VERIFIER_DETECTED_VIOLATION).

Without any arguments, the **!deadlock** extension causes the basic lock hierarchy topology to be displayed. If the problem is not a simple cyclical deadlock, this command will describe the situation that has occurred.

The **!deadlock 1** extension causes stack traces to be displayed. The stacks displayed will be the ones active at the time the locks were acquired.

Here is an example:

```
dbgcmd
0:kd> !deadlock

Deadlock detected (2 resources in 2 threads):

Thread 0: A B
Thread 1: B A
```

Where:

```
Thread 0 = 8d3ba030
Thread 1 = 8d15c030
Lock A = bba2af30 Type 'Spinlock'
Lock B = dummy!GlobalLock Type 'Spinlock'
```

This tells you which threads and which locks are involved. However, it is intended to be a summary and may not be enough information to adequately debug the situation.

Use **!deadlock 1** to print out the contents of the call stacks at the time that each lock participating in the deadlock was acquired. Because these are run-time stack traces, they will be more complete if a checked build is being used. Checked builds were available on older versions of Windows before Windows 10, version 1803. On a free build, they may be truncated after as little as one line.

```
dbgcmd
```

```
0:kd> !deadlock 1
```

```
Deadlock detected (2 resources in 2 threads):
```

```
Thread 0 (8D14F750) took locks in the following order:
```

```
Lock A -- b7906f30 (Spinlock)
Stack:    dummy!DummyActivateVcComplete+0x63
          dummy!dummyOpenVcChannels+0x2E1
          dummy!DummyAllocateRecvBufferComplete+0x436
          dummy!DummyAllocateComplete+0x55
          NDIS!ndisMQueuedAllocateSharedHandler+0xC9
          NDIS!ndisWorkerThread+0xEE
```

```
Lock B -- dummy!GlobalLock (Spinlock)
Stack:    dummy!dummyQueueRecvBuffers+0x2D
          dummy!DummyActivateVcComplete+0x90
          dummy!dummyOpenVcChannels+0x2E1
          dummy!DummyAllocateRecvBufferComplete+0x436
          dummy!DummyAllocateComplete+0x55
```

```
Thread 1 (8D903030) took locks in the following order:
```

```
Lock B -- dummy!GlobalLock (Spinlock)
Stack:    dummy!dummyRxInterruptOnCompletion+0x25D
          dummy!DummyHandleInterrupt+0x32F
          NDIS!ndisMDpcX+0x3C
          ntkrnlp!KiRetireDpcList+0x5D
```

```
Lock A -- b7906f30 (Spinlock)
Stack:    << Current stack >>
```

With this information, you have almost everything you need, except the current stack:

```
dbgcmd

0: kd> k
ChildEBP RetAddr
f78aae6c 80664c58 ntkrn1pa!DbgBreakPoint
f78aae74 8066523f ntkrn1pa!ViDeadlockReportIssue+0x2f
f78aae9c 806665df ntkrn1pa!ViDeadlockAnalyze+0x253
f78aaee8 8065d944 ntkrn1pa!VfDeadlockAcquireResource+0x20b
f78aaf08 bfd6df46 ntkrn1pa!VerifierKeAcquireSpinLockAtDpcLevel+0x44
f78aafa4 b1bf2d2d dummy!dummyRxInterruptOnCompletion+0x2b5
f78aafc4 bfde9d8c dummy!DummyHandleInterrupt+0x32f
f78aafd8 804b393b NDIS!ndisMDpcX+0x3c
f78aaff4 804b922b ntkrn1pa!KiRetireDpcList+0x5d
```

From this you can see which locks were involved and where they were acquired. This should be enough information for you to debug the deadlock. If the source code is available, you can use the debugger to see exactly where the problem occurred:

```
dbgcmd

0: kd> .lines
Line number information will be loaded

0: kd> u dummy!DummyActivateVcComplete+0x63 11
dummy!DummyActivateVcComplete+63 [d:\nt\drivers\dummy\vc.c @ 2711]:
b1bfe6c9 837d0c00      cmp     dword ptr [ebp+0xc],0x0

0: kd> u dummy!dummyQueueRecvBuffers+0x2D 11
dummy!dummyQueueRecvBuffers+2d [d:\nt\drivers\dummy\receive.c @ 2894]:
b1bf4e39 807d0c01      cmp     byte ptr [ebp+0xc],0x1

0: kd> u dummy!dummyRxInterruptOnCompletion+0x25D 11
dummy!dummyRxInterruptOnCompletion+25d [d:\nt\drivers\dummy\receive.c @
1424]:
b1bf5d05 85f6          test    esi,esi

0: kd> u dummy!dummyRxInterruptOnCompletion+0x2b5 11
dummy!dummyRxInterruptOnCompletion+2b5 [d:\nt\drivers\dummy\receive.c @
1441]:
b1bf5d5d 8b4648        mov     eax,[esi+0x48]
```

Now you know the name of the source file and the line number where the acquisition took place. In this case, the source files will show that the threads behaved as follows:

- Thread 1: **DummyActivateVcComplete** took the **dummy** miniport lock. It then called **dummyQueueRecvBuffers**, which took the **dummy** global lock.

- Thread 2: `dummyRxInterruptOnCompletion` took the global lock. Then, a few lines later, it took the miniport lock.

At this point, the deadlock becomes entirely clear.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!defwrites

Article • 10/25/2023

The **!defwrites** extension displays the values of the kernel variables used by the cache manager.

```
dbgcmd
```

```
!defwrites
```

DLL

Kdexts.dll

Additional Information

For information about write throttling, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, use the [!cchelp](#) extension.

Remarks

When the number of deferred writes ("dirty pages") becomes too large, page writing will be throttled. This extension allows you to see whether your system has reached this point.

Here is an example:

```
dbgcmd
```

```
kd> !defwrites
*** Cache Write Throttle Analysis ***
```

CcTotalDirtyPages:	0 (0 Kb)
CcDirtyPageThreshold:	1538 (6152 Kb)
MmAvailablePages:	2598 (10392 Kb)
MmThrottleTop:	250 (1000 Kb)
MmThrottleBottom:	30 (120 Kb)
MmModifiedPageListHead.Total:	699 (2796 Kb)

```
Write throttles not engaged
```

In this case, there are no dirty pages. If **CcTotalDirtyPages** reaches 1538 (the value of **CcDirtyPageThreshold**), writing will be delayed until the number of dirty pages is reduced.

!devext

Article • 10/25/2023

The **!devext** extension displays bus-specific device extension information for devices on a variety of buses.

```
dbgcmd
```

```
!devext Address TypeCode
```

Parameters

Address

Specifies the hexadecimal address of the device extension to be displayed.

TypeCode

Specifies the type of object that owns the device extension to be displayed. Type codes are not case-sensitive. Valid type codes are:

TypeCode	Object
ISAPNP	ISA PnP device extension
PCMCIA	PCMCIA device extension
HID	HID device extension

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For more information about device extensions, see the Windows Driver Kit (WDK) documentation.

Remarks

The **!usbhub**, **!hidfdo**, and **!hidpdo** extensions are obsolete; their functionality has been integrated into **!devext**.

For those object types that are no longer supported by **!devext**, use the [dt \(Display Type\)](#) debugger command.

Here is an example for an ISA PnP device extension:

```
dbgcmd

kd> !devext e0000165fff32190 ISAPNP
ISA PnP FDO @ 0x00000000, DevExt @ 0xe0000165fff32190, Bus # 196639
Flags (0x854e2530)  DF_ACTIVATED, DF_QUERY_STOPPED,
                     DF_STOPPED, DF_RESTARTED_NOMOVE,
                     DF_BUS
                     Unknown flags 0x054e2000

NumberCSNs           - -536870912
ReadDataPort         - 0x0000000d (mapped)
AddressPort          - 0x00000000 (not mapped)
CommandPort          - 0x00000000 (not mapped)
DeviceList           - 0xe000000085007b50
CardList              - 0x00000000
PhysicalBusDevice   - 0x00000000
AttachedDevice        - 0x00000000
SystemPowerState     - Unspecified
DevicePowerState     - Unspecified
```

Here is an example for a PCI device:

```
dbgcmd

kd> !devext e0000000858c31b0 PCI
PDO Extension, Bus 0x0, Device 0, Function 0.
DevObj 0xe0000000858c3060 PCI Parent Bus FDO DevExt 0xe0000000858c4960
Device State = PciNotStarted
Vendor ID 8086 (INTEL) Device ID 123D
Class Base/Sub 08/00 (Base System Device/Interrupt Controller)
Programming Interface: 20, Revision: 01, IntPin: 00, Line Raw/Adj 00/00
Enables ((cmd & 7) = 106): BM Capabilities Pointer = <none>
CurrentState:       System Working, Device D0
WakeLevel:          System Unspecified, Device Unspecified
Requirements: <none>
```

!devhandles

Article • 04/03/2024

The **!devhandles** extension displays the open handles for the specified device.

```
dbgcmd
```

```
!devhandles Address
```

Parameters

Address

Specifies the address of the device for which to display the open handles.

DLL

Kdexts.dll

Remarks

To display complete handle information, this extension requires private symbols.

The address of a device object can be obtained using the [!drvobj](#) or [!devnode](#) extensions.

Here is a truncated example:

```
dbgcmd
```

```
1kd> !devhandles 0x841153d8
```

```
Checking handle table for process 0x840d3940
Handle table at 95fea000 with 578 Entries in use
```

```
Checking handle table for process 0x86951d90
Handle table at 8a8ef000 with 28 Entries in use
```

```
...
```

```
Checking handle table for process 0x87e63650
Handle table at 947bc000 with 308 Entries in use
```

```
Checking handle table for process 0x87e6f4f0
00000000: Unable to read handle table
```

Feedback

Was this page helpful?

 Yes

 No

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

!devnode

Article • 10/25/2023

The **!devnode** extension displays information about a node in the device tree.

```
dbgcmd  
!devnode Address [Flags] [Service]  
!devnode 1  
!devnode 2
```

Parameters

Address

Specifies the hexadecimal address of the device extension whose node is to be displayed. If this is zero, the root of the main device tree is displayed.

Flags

Specifies the level of output to be displayed. This can be any combination of the following bits:

Bit 0 (0x1)

Causes the display to include all children of the device node.

Bit 1 (0x2)

Causes the display to include resources used (CM_RESOURCE_LIST). These include the boot configuration reported by IRP_MN_QUERY_RESOURCES, as well as the resources allocated to the device in the *AllocatedResources* parameter of IRP_MN_START_DEVICE.

Bit 2 (0x4)

Causes the display to include resources required (IO_RESOURCE_REQUIREMENTS_LIST) as reported by IRP_MN_FILTER_RESOURCE_REQUIREMENTS.

Bit 3 (0x8)

Causes the display to include a list of translated resources as allocated to the device in the *AllocatedResourcesTranslated* parameter of IRP_MN_START_DEVICE.

Bit 4 (0x10)

Specifies that only device nodes that are not started should be displayed.

Bit 5 (0x20)

Specifies that only device nodes with problems should be displayed. (These are nodes

that contain the flag bits DNF_HAS_PROBLEM or DNF_HAS_PRIVATE_PROBLEM.)

Service

Specifies the name of a service. If this is included, only those device nodes driven by this service will be displayed. (If *Flags* includes bit 0x1, device nodes driven by this service and all their children will be displayed.)

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about device trees, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!devnode 1** command lists all pending removals of device objects.

The **!devnode 2** command lists all pending ejects of device objects.

You can use **!devnode 0 1** to see the entire device tree.

!devobj

Article • 10/25/2023

The **!devobj** extension displays detailed information about a DEVICE_OBJECT structure.

```
dbgcmd
```

```
!devobj DeviceObject
```

Parameters

DeviceObject

Specifies the device object. This can be the hexadecimal address of this structure or the name of the device.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for examples and applications of this extension command. For information about device objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If *DeviceObject* specifies the name of the device but supplies no prefix, the prefix "\Device\" is assumed. Note that this command will check to see if *DeviceObject* is a valid address or device name before using the expression evaluator.

The information displayed includes the device name of the object, information about the device's current IRP, and a list of addresses of any pending IRPs in the device's queue. It also includes information about device objects layered on top of this object (listed as "AttachedDevice") and those layered under this object (listed as "AttachedTo").

The address of a device object can be obtained using the [!drvobj](#) or [!devnode](#) extensions.

Here is one example:

dbgcmd

```
kd> !devnode
Dumping IopRootDeviceNode (= 0x80e203b8)
DevNode 0x80e203b8 for PDO 0x80e204f8
Parent 0000000000 Sibling 0000000000 Child 0x80e56dc8
InstancePath is "HTREE\ROOT\0"
State = DeviceNodeStarted (0x308)
Previous State = DeviceNodeEnumerateCompletion (0x30d)
StateHistory[04] = DeviceNodeEnumerateCompletion (0x30d)
StateHistory[03] = DeviceNodeStarted (0x308)
StateHistory[02] = DeviceNodeEnumerateCompletion (0x30d)
StateHistory[01] = DeviceNodeStarted (0x308)
StateHistory[00] = DeviceNodeUninitialized (0x301)
StateHistory[19] = Unknown State (0x0)
.....
StateHistory[05] = Unknown State (0x0)
Flags (0x00000131) DNF_MADEUP, DNF_ENUMERATED,
DNF_IDS_QUERIED, DNF_NO_RESOURCE_REQUIRED
DisableableDepends = 11 (from children)

kd> !devobj 80e204f8
Device object (80e204f8) is for:
\Driver\PnpManager DriverObject 80e20610
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001000
DevExt 80e205b0 DevObjExt 80e205b8 DevNode 80e203b8
ExtensionFlags (0000000000)
Device queue is not busy.
```

!devstack

Article • 10/25/2023

The **!devstack** extension displays a formatted view of the device stack associated with a device object.

```
dbgcmd
```

```
!devstack DeviceObject
```

Parameters

DeviceObject

Specifies the device object. This can be the hexadecimal address of the DEVICE_OBJECT structure or the name of the device.

DLL

Kdexts.dll

Additional Information

For information about device stacks, see the Windows Driver Kit (WDK) documentation.

Remarks

If *DeviceObject* specifies the name of the device but supplies no prefix, the prefix "\Device\" is assumed. Note that this command will check to see if *DeviceObject* is a valid address or device name before using the expression evaluator.

Here is an example:

```
dbgcmd
```

```
kd> !devstack e000000085007b50
!DevObj   !DrvObj          !DevExt   ObjectName
e0000165fff32040  \Driver\kmixer    e0000165fff32190
> e000000085007b50  \Driver\swenum    e000000085007ca0  KSENUM#00000005
!DevNode e0000165fff2e010 :
DeviceInst is "SW\{b7eafdc0-a680-11d0-96d8-00aa0051e51d}\{9B365890-165F-
```

11D0-A195-0020AFD156E4}"
ServiceName is "kmixer"

!dflink

Article • 10/25/2023

The **!dflink** extension displays a linked list in the forward direction.

dbgcmd

```
!dflink Address [Count] [Bias]
```

Parameters

Address

Specifies the address of a LIST_ENTRY structure. The display will begin with this node.

Count

Specifies the maximum number of list entries to display. If this is omitted, the default is 32.

Bias

Specifies a mask of bits to ignore in each pointer. Each **Flink** address is ANDed with (NOT *Bias*) before following it to the next location. The default is zero (in other words, do not ignore any bits).

DLL

Kdexts.dll

Remarks

The **!dflink** extension traverses the **Flink** fields of the LIST_ENTRY structure and displays up to four ULONGs at each address. To go in the other direction, use [!dblink](#).

The [dl \(Display Linked List\)](#) command is more versatile than [!dblink](#) and [!dflink](#).

!diskspace

Article • 10/25/2023

The **!diskspace** extension displays the amount of free space on a hard disk of the target computer.

```
dbgcmd
```

```
!diskspace Drive[:]
```

Parameters

Drive

Specifies the drive letter of the disk. The colon (:) after *Drive* is optional.

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

Remarks

Here is an example:

```
dbgcmd
```

```
kd> !diskspace c:  
Checking Free Space for c: .....  
Cluster Size 0 KB  
Total Clusters 4192901 KB  
Free Clusters 1350795 KB  
Total Space 1 GB (2096450 KB)  
Free Space 659.567871 MB (675397 KB)
```

```
kd> !diskspace f:  
Checking Free Space for f:  
f: is a CDROM drive. This function is not supported!
```

!dma

Article • 04/03/2024

The **!dma** extension displays information about the Direct Memory Access (DMA) subsystem, and the **DMA Verifier** option of Driver Verifier.

```
dbgcmd
```

```
!dma  
!dma Adapter [Flags]
```

Parameters

Adapter

Specifies the hexadecimal address of the DMA adapter to be displayed. If this is zero, all DMA adapters will be displayed.

Flags

Specifies the information to include in the display. This can be any combination of the following bits. The default is 0x1.

Bit 0 (0x1)

Causes the display to include generic adapter information.

Bit 1 (0x2)

Causes the display to include map register information. (Only when DMA Verification is active.)

Bit 2 (0x4)

Causes the display to include common buffer information. (Only when DMA Verification is active.)

Bit 3 (0x8)

Causes the display to include scatter/gather list information. (Only when DMA Verification is active.)

Bit 4 (0x10)

Causes the display to include the device description for the hardware device. (Only when DMA Verification is active.)

Bit 5 (0x20)

Causes the display to include Wait context block information.

DLL

Kdexts.dll

Additional Information

For information about Driver Verifier, see the Windows Driver Kit (WDK) documentation.

For information about DMA, see the Windows Driver Kit (WDK) documentation and

Microsoft Windows Internals by Mark Russinovich David Solomon.

Remarks

Invalid arguments (for example, **!dma 1**) generate a brief help text.

When the **!dma** extension is used with no parameters, it displays a concise list of all DMA adapters and their addresses. This can be used to obtain the address of an adapter for use in the longer versions of this command.

Here is an example of how this extension can be used when the Driver Verifier's **DMA Verification** option is active:

```
dbgcmd
0:kd> !dma
Dumping all DMA adapters...
Adapter: 82faebd0      Owner: SCSIPORT!ScsiPortGetUncachedExtension
Adapter: 82f88930      Owner: SCSIPORT!ScsiPortGetUncachedExtension
Adapter: 82f06cd0      Owner: NDIS!NdisMAllocateMapRegisters
Master adapter: 80076800
```

From this output, you can see that there are three DMA adapters in the system. SCSIPORT owns two and NDIS owns the third. To examine the NDIS adapter in detail, use the **!dma** extension with its address:

```
dbgcmd
0:kd> !dma 82f06cd0
Adapter: 82f06cd0      Owner: NDIS!NdisMAllocateMapRegisters (0x9fe24351)
MasterAdapter:          00000000
Adapter base Va        00000000
Map register base:     00000000
WCB:                  82f2b604
Map registers: 00000000 mapped, 00000000 allocated, 00000002 max
```

```
Dma verifier additional information:  
DeviceObject: 82f98690  
Map registers: 00000840 allocated, 00000000 freed  
Scatter-gather lists: 00000000 allocated, 00000000 freed  
Common buffers: 00000004 allocated, 00000000 freed  
Adapter channels: 00000420 allocated, 00000420 freed  
Bytes mapped since last flush: 000000f2
```

The first block of data is specific information that a HAL developer can use to debug the problem. For your purposes, the data below "Dma verifier additional information" is what is interesting. In this example, you see that NDIS has allocated 0x840 map registers. This is a huge number, especially because NDIS had indicated that it planned to use a maximum of two map registers. This adapter apparently does not use scatter/gather lists and has put away all its adapter channels. Look at the map registers in more detail:

```
dbgcmd
```

```
0:kd> !dma 82f06cd0 2  
Adapter: 82f06cd0 Owner: NDIS!NdisAllocateMapRegisters  
...  
  
Map register file 82f06c58 (0/2 mapped)  
Double buffer mdl: 82f2c188  
Map registers:  
    82f06c80: Not mapped  
    82f06c8c: Not mapped  
  
Map register file 82f06228 (1/2 mapped)  
Double buffer mdl: 82f1b678  
Map registers:  
    82f06250: 00bc bytes mapped to f83c003c  
    82f0625c: Not mapped  
  
Map register file 82fa5ad8 (1/2 mapped)  
Double buffer mdl: 82f1b048  
Map registers:  
    82fa5b00: 0036 bytes mapped to 82d17102  
    82fa5b0c: Not mapped  
...
```

In this example, you see that certain map registers have been mapped. Each *map register file* is an allocation of map registers by the driver. In other words, it represents a single call to **AllocateAdapterChannel**. NDIS collects a large number of these map register files, while some drivers create them one at a time and dispose of them when they are finished.

The map register files are also the addresses that are returned to the device under the name "MapRegisterBase". Note that DMA verifier only hooks the first 64 map registers for each driver. The rest are ignored for reasons of space (each map register represents three physical pages).

In this example, two map register files are in use. This means that the driver has mapped a buffer so it is visible to the hardware. In the first case, 0xBC bytes are mapped to the system virtual address 0xF83C003C.

An examination of the common buffers reveals:

```
dbgcmd

0:kd> !dma 82f06cd0 4
Adapter: 82f06cd0      Owner: NDIS!NdisMAllocateMapRegisters
...
Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
Length:          1000
Virtual address: 82e77000
Physical address: 2a77000

Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
Length:          12010
Virtual address: 82e817f8
Physical address: 2a817f8

Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
Length:          4300
Virtual address: 82e95680
Physical address: 2a95680

Common buffer allocated by NDIS!NdisMAllocateSharedMemory:
Length:          4800
Virtual address: 82e9d400
Physical address: 2a9d400
```

This is fairly straightforward; there are four common buffers of varying lengths. The physical and virtual addresses are all given.

Feedback

Was this page helpful?

 Yes

 No

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

!dpa

Article • 04/03/2024

The **!dpa** extension displays pool allocation information.

```
dbgcmd  
  !dpa Options  
  !dpa -?
```

Parameters

Options Must be exactly one of the following options:

-c

Displays current pool allocation statistics.

-v

Displays all current pool allocations.

-vs

Causes the display to include stack traces.

-f

Displays failed pool allocations.

-r

Displays free pool allocations.

-p ** Ptr**

Displays all pool allocations that contain the pointer *Ptr*.

-?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Kdexts.dll

Remarks

Pool instrumentation must be enabled in Win32k.sys in order for this extension to work.

Feedback

Was this page helpful?

 Yes

 No

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

!dpcs

Article • 10/25/2023

The **!dpcs** extension displays the deferred procedure call (DPC) queues for a specified processor.

dbgcmd

```
!dpcs [Processor]
```

Parameters

Processor

Specifies a processor. If *Processor* is omitted, then the DPC queues for all processors are displayed.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003 and later	Kdexts.dll

Additional Information

For information about DPCs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!driveinfo

Article • 05/30/2024

The **!driveinfo** extension displays volume information for the specified drive.

```
dbgcmd
```

```
!driveinfo Drive[:]
!driveinfo
```

Parameters

Drive

Specifies a drive. The colon at the end of the drive name is optional.

No parameters

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Kdexts.dll

Remarks

The drive information displayed by this extension is obtained by querying the underlying file system; for example:

```
dbgcmd
```

```
kd> !driveinfo c:
Drive c:, DriveObject e136cd88
    Directory Object: e1001408  Name: C:
        Target String is '\Device\HarddiskVolume1'
        Drive Letter Index is 3 (C:)
    Volume DevObj: 82254a68
        Vpb: 822549e0  DeviceObject: 82270718
    FileSystem: \FileSystem\Ntfs
        Volume has 0x229236 (free) / 0x2ee1a7 (total) clusters of size 0x1000
        8850.21 of 12001.7 MB free
```

Feedback

Was this page helpful?

 Yes

 No

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

!drivers

Article • 10/25/2023

ⓘ Note

In Windows XP and later versions of Windows, the **!drivers** extension is obsolete. To display information about loaded drivers and other modules, use the **!m** command.

The command **!m t n** displays information in a format very similar to the old **!drivers** extension. However, this command will not display the memory usage of the drivers as the **!drivers** extension did. It will only display the drivers' start and end addresses, image names, and timestamps. The **!vm** and **!memusage** extensions can be used to display memory usage statistics.

```
dbgcmd
```

```
!drivers [Flags]
```

Parameters

Flags

Can be any combination of the following values. (The default is 0x0.)

Bit 0 (0x1)

Causes the display to include information about resident and standby memory.

Bit 1 (0x2)

If this bit is set and bit 2 (0x4) is not set, the display will include information about resident, standby, and locked memory, as well as the loader entry address. If bit 2 is set, this causes the display to be a much longer and more detailed list of the driver image. Information about the headers is included, as is section information.

Bit 2 (0x4)

Causes the display to be a much longer and more detailed list of the driver image. Information about each section is included. If bit 1 (0x2) is set, this will also include header information.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about drivers and their memory use, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

An explanation of this command's display is given in the following table:

Column	Meaning
Base	The starting address of the device driver code, in hexadecimal. When the memory address used by the code that causes a stop falls between the base address for a driver and the base address for the next driver in the list, that driver is frequently the cause of the fault. For instance, the base for Ncrc810.sys is 0x80654000. Any address between that and 0x8065a000 belongs to this driver.
Code Size	The size, in kilobytes, of the driver code, in both hexadecimal and decimal.
Data Size	The amount of space, in kilobytes, allocated to the driver for data, in both hexadecimal and decimal.
Locked	(Only when Flag 0x2 is used) The amount of memory locked by the driver.
Resident	(Only when Flag 0x1 or 0x2 is used) The amount of the driver's memory that actually resides in physical memory.
Standby	(Only when Flag 0x1 or 0x2 is used) The amount of the driver's memory that is on standby.
Loader Entry	(Only when Flag 0x2 is used) The loader entry address.

Column	Meaning
Driver Name	The driver file name.
Creation Time	The link date of the driver. Do not confuse this with the file date of the driver, which can be set by external tools. The link date is set by the compiler when a driver or executable file is compiled. It should be close to the file date, but it is not always the same.

The following is a truncated example of this command:

```
dbgcmd

kd> !drivers
Loaded System Driver Summary
Base      Code Size      Data Size      Driver Name Creation Time
80080000 f76c0 (989 kb) 1f100 (124 kb) ntoskrnl.exe Fri May 26 15:13:00
80400000 d980  ( 54 kb) 4040  ( 16 kb) hal.dll      Tue May 16 16:50:34
80654000 3f00  ( 15 kb) 1060  ( 4 kb) ncrc810.sys Fri May 05 20:07:04
8065a000 a460  ( 41 kb) 1e80  ( 7 kb) SCSIPORT.SYS Fri May 05 20:08:05
```

!drvobj

Article • 10/25/2023

The `!drvobj` extension displays detailed information about a DRIVER_OBJECT.

dbgcmd

```
!drvobj DriverObject [Flags]
```

Parameters

DriverObject

Specifies the driver object. This can be the hexadecimal address of the DRIVER_OBJECT structure or the name of the driver.

Flags

Can be any combination of the following bits. (The default is 0x01.)

Bit 0 (0x1)

Causes the display to include device objects owned by the driver.

Bit 1 (0x2)

Causes the display to include entry points for the driver's dispatch routines.

Bit 2 (0x4)

Lists with detailed information the device objects owned by the driver (requires bit 0 (0x1)).

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for examples and applications of this extension command.

For information about driver objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If *DriverObject* specifies the name of the device but supplies no prefix, the prefix "\Driver\" is assumed. Note that this command will check to see if *DriverObject* is a valid address or device name before using the expression evaluator.

If *DriverObject* is an address, it must be the address of the DRIVER_OBJECT structure. This can be obtained by examining the arguments passed to the driver's **DriverEntry** routine.

This extension command will display a list of all device objects created by a specified driver. It will also display all fast I/O routines registered with this driver object.

The following is an example for the Symbios Logic 810 SCSI miniport driver:

```
dbgcmd

kd> bp DriverEntry          // breakpoint at DriverEntry

kd> g
symc810!DriverEntry+0x40:
80006a20: b07e0050 stl      t2,50(sp)

kd> r a0 //address of DevObj (the first parameter)
a0=809d5550

kd> !drvobj 809d5550 // display the driver object
Driver object is for:
\Driver\symc810
Device Object list:
809d50d0
```

You can also use [!devobj 809d50d0](#) to get information about the device object.

!dskheap

Article • 04/03/2024

The **!dskheap** extension displays desktop heap information for a specified session.

```
dbgcmd
```

```
!dskheap [-v] [-s SessionID]
```

Parameters

-v

Causes the display to include more detailed output.

-s ** SessionID**

Specifies a session. If this parameter is omitted, then the desktop heap information for session 0 is displayed.

DLL

Kdexts.dll

Additional Information

For information about desktops or desktop heaps, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The desktop heap information for the session is arranged by window station.

Here are a couple of examples:

```
dbgcmd
```

```
kd> !dskheap -s 3
##   Winstation\Desktop          Heap Size(KB)    Used Rate(%)
WinSta0\Screen-saver             3072            0%
WinSta0\Default                  3072            0%
```

```
WinSta0\Disconnect          64           4%
##  WinSta0\Winlogon        128          5%

Total Desktop: (      6336 KB -   4 desktops)
#           Session ID: 3

kd> !dskheap
##  Winstation\Desktop      Heap Size(KB)  Used Rate(%)

WinSta0\Default            3072          0%
WinSta0\Disconnect         64           4%
WinSta0\Winlogon           128          9%
Service-0x0-3e7$\Default  512           4%
Service-0x0-3e5$\Default  512          0%
Service-0x0-3e4$\Default  512           1%
##  SAWinSta\SADesktop       512          0%

Total Desktop: (      5312 KB -   7 desktops)
#           Session ID: 0
```

Feedback

Was this page helpful?

 Yes

 No

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

!eb, !ed

Article • 04/03/2024

The **!eb** and **!ed** extensions write a sequence of values into a specified physical address.

These extension commands should not be confused with the [e* \(Enter Values\)](#) command.

dbgcmd

```
!eb [Flag] PhysicalAddress Data [ ... ]  
!ed [Flag] PhysicalAddress Data [ ... ]
```

Parameters

Flag

Can be any one of the following values. The *Flag* value must be surrounded by square brackets:

[c]

Writes to cached memory.

[uc]

Writes to uncached memory.

[wc]

Writes to write-combined memory.

PhysicalAddress

Specifies the first physical address on the target computer to which the data will be written, in hexadecimal.

Data

Specifies one or more values to be written sequentially into physical memory. Enter these values in hexadecimal format. For the **!eb** extension, each value must be 1 byte (two hexadecimal digits). For the **!ed** extension, each value must be one DWORD (eight hexadecimal digits). You can include any number of *Data* values on one line. To separate multiple values, use commas or spaces.

DLL

Additional Information

To read physical memory, use the `!d*` extensions. For an overview of memory manipulation and a description of other memory-related commands, see [Reading and Writing Memory](#).

Feedback

Was this page helpful?

 Yes

 No

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

!ecb, !ecd, !ecw

Article • 10/25/2023

The **!ecb**, **!ecd**, and **!ecw** extensions write to the PCI configuration space.

dbgcmd

!ec Bus.Device.Function Offset Data

Parameters

Bus

Specifies the bus. *Bus* can range from 0 to 0xFF.

Device

Specifies the slot device number for the device.

Function

Specifies the slot function number for the device.

Offset

Specifies the address at which to write.

Data

Specifies the value to be written. For the **!ecb** extension, *Data* must be 1 byte (two hexadecimal digits). For the **!ecw** extension, *Data* must be one WORD (four hexadecimal digits). For the **!ecd** extension, *Data* must be one DWORD (eight hexadecimal digits).

DLL

Windows 2000	Kext.dll
Windows XP and later	Kext.dll

These extension commands can only be used with an x86-based target computer.

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command, and some additional examples. For information about PCI buses, see the Windows Driver Kit (WDK) documentation.

Remarks

You cannot use these extension commands to write a sequence of *Data* values. This can only be done by repeated use of this extension.

To display the PCI configuration space, use [**!pci 100Bus Device Function**](#).

!ecs

Article • 10/25/2023

The **!ecs** extension is obsolete. To edit the PCI configuration space, use [**!ecb**](#), [**!ecd**](#), or [**!ecw**](#).

!errlog

Article • 10/25/2023

The **!errlog** extension displays the contents of any pending entries in the I/O system's error log.

```
dbgcmd
```

```
!errlog
```

DLL

Kdexts.dll

Additional Information

For information about [IoWriteErrorLogEntry](#), see the Windows Driver Kit (WDK) documentation.

Remarks

This command displays information about any pending events in the I/O system's error log. These are events queued by calls to the [IoWriteErrorLogEntry](#) function, to be written to the system's event log for subsequent viewing by the [Event Viewer](#).

Only entries that were queued by [IoWriteErrorLogEntry](#) but have not been committed to the error log will be displayed.

This command can be used as a diagnostic aid after a system crash because it reveals pending error information that was unable to be committed to the error log before the system halted.

!errpkt

Article • 10/25/2023

The **!errpkt** extension displays the contents of a Windows Hardware Error Architecture (WHEA) hardware error packet.

```
dbgcmd
```

```
!errpkt Address
```

Parameters

Address

Specifies the address of the hardware error packet.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003	Unavailable
Windows Vista and later	Kdexts.dll

This extension can be used only in Windows Vista and later versions of Windows.

Additional Information

The **!whea** and **!errrec** extensions can be used to display additional WHEA information.

For general information about WHEA, see [Windows Hardware Error Architecture \(WHEA\)](#) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows the output of the **!errpkt** extension:

```
dbgcmd
```

```
3: kd> !errpkt ffffffa8007cf44da
WHEA Error Packet Info Section (@ ffffffa8007cf44da)
```

```
Flags          : 0x00000000
Size           : 0x218
RawDataLength  : 0x392
Context         : 0x0000000000000000
ErrorType       : 0x0 - Processor
ErrorSeverity   : 0x1 - Fatal
ErrorSourceId   : 0x0
ErrorSourceType : 0x0 - MCE
Version         : 00000002
Cpu             : 0000000000000002
RawDataFormat   : 0x2 - Intel64 MCA

Raw Data        : Located @ FFFFFA8007CF45F2
```

Processor Error: (Internal processor error)

This error means either the processor is damaged or perhaps voltage and/or temperature thresholds have been exceeded.
If the problem continues to occur, replace the processor.

Processor Number : 2

Bank Number : 0

Status : 0

Address : 0000000000000000 (I)

Misc : 0000000000000000 (I)

!errrec

Article • 10/25/2023

The **!errrec** extension displays the contents of a Windows Hardware Error Architecture (WHEA) error record.

```
dbgcmd
```

```
!errrec Address
```

Parameters

Address

Specifies the address of the error record.

DLL

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003	Unavailable
Windows Vista and later	Kdexts.dll

This extension can be used only in Windows Vista and later versions of Windows.

Additional Information

The [!whea](#) and [!errpkt](#) extensions can be used to display additional WHEA information. For general information about WHEA, see [Windows Hardware Error Architecture \(WHEA\)](#) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows how the [!whea](#) extension can be used to obtain the address of an error record, and then the contents of this record can be displayed by the **!errrec** extension:

```
dbgcmd
```

```
3: kd> !whea
Error Source Table @ ffffff800019ca250
13 Error Sources
Error Source 0 @ ffffffa80064132c0
    Notify Type      : Machine Check Exception
    Type            : 0x0 (MCE)
    Error Count     : 8
    Record Count    : 10
    Record Length   : c3e
    Error Records   : wrapper @ ffffffa8007cf4000 record @ ffffffa8007cf4030

    . . .

# 3: kd> !errrec ffffffa8007cf4030
=====
===
## Common Platform Error Record @ ffffffa8007cf4030
-----
---

Revision      : 2.1
Record Id     : 01c9c7ff04e0ff7d
Severity      : Fatal (1)
Length        : 1730
Creator       : Microsoft
Notify Type   : Machine Check Exception
Timestamp     : 4/28/2009 12:54:47
Flags         : 0x00000000
#
=====

===
## Section 0      : Processor Generic
-----
---

Descriptor    @ ffffffa8007cf40b0
Section       @ ffffffa8007cf4188
Offset        : 344
Length        : 192
Flags         : 0x00000001 Primary
Severity      : Fatal

No valid data fields are present.
#
=====

===
## Section 1      : {390f56d5-ca86-4649-95c4-73a408ae5834}
-----
---

Descriptor    @ ffffffa8007cf40f8
Section       @ ffffffa8007cf4248
Offset        : 536
Length        : 658
Flags         : 0x00000000
Severity      : Fatal
```

```
*** Unknown section format ***
#
=====
===
## Section 2      : Error Packet
-----
---

Descriptor      @ ffffffa8007cf4140
Section         @ ffffffa8007cf44da
Offset          : 1194
Length          : 536
Flags           : 0x00000000
Severity        : Fatal

WHEA Error Packet Info Section (@ ffffffa8007cf44da)
Flags           : 0x00000000
Size            : 0x218
RawDataLength   : 0x392
Context         : 0x0000000000000000
ErrorType       : 0x0 - Processor
ErrorSeverity   : 0x1 - Fatal
ErrorSourceId   : 0x0
ErrorSourceType : 0x0 - MCE
Version         : 00000002
Cpu             : 0000000000000002
RawDataFormat   : 0x2 - Intel64 MCA

Raw Data        : Located @ FFFFFA8007CF45F2

Processor Error: (Internal processor error)
This error means either the processor is damaged or perhaps
voltage and/or temperature thresholds have been exceeded.
If the problem continues to occur, replace the processor.
Processor Number : 2
Bank Number     : 0
Status          : 0
Address         : 0000000000000000 (I)
Misc            : 0000000000000000 (I)
```

!exca

Article • 04/03/2024

The **!exca** extension displays PC-Card Interrupt Controller (PCIC) Exchangable Card Architecture (ExCA) registers.

```
dbgcmd
```

```
!exca BasePort.SocketNumber
```

Parameters

BasePort

Specifies the base port of the PCIC.

SocketNumber

Specifies the socket number of the ExCA register on the PCIC.

DLL

Kext.dll

The **!exca** extension is only available for an x86-based target computer.

Additional Information

The [!cbreg](#) extension can be used to display CardBus Socket registers and CardBus ExCA registers by address.

Feedback

Was this page helpful?

 Yes

 No

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

!filecache

Article • 10/25/2023

The **!filecache** extension displays information regarding the system file cache memory and PTE use.

```
dbgcmd
```

```
!filecache [Flags]
```

Parameters

Flags

Optional. Default value is 0x0. Set *Flags* to 0x1 to sort the output by shared cache map. This way you can locate all the system cache views for a given control area.

DLL

Kdexts.dll

Additional Information

For information about file system drivers, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Each line of this extension's output represents a virtual address control block (VACB). When named files are mapped into the VACB, the names of these files are displayed. If "no name for file" is specified, this means that this VACB is being used to cache metadata.

Here is an example of the output from this extension from a Windows XP system:

```
dbgcmd
```

```
kd> !filecache
***** Dump file cache*****
Reading and sorting VACBs ...
```

```
Removed 1811 nonactive VACBs, processing 235 active VACBs ...
File Cache Information
  Current size 28256 kb
  Peak size    30624 kb
  235 Control Areas
Skipping view @ c1040000 - no VACB, but PTE is valid!
  Loading file cache database (100% of 131072 PTEs)
  SkippedPageTableReads = 44
  File cache has 4056 valid pages
```

Usage Summary (in Kb):

Control	Valid	Standby/Dirty	Shared	Locked	Name
817da668	4	0	0	0	\$MftMirr
8177ae68	304	920	0	0	\$LogFile
81776160	188	0	0	0	\$BitMap
817cf370	4	0	0	0	\$Mft
81776a00	8	0	0	0	\$Directory
817cfdd0	4	0	0	0	\$Directory
81776740	36	0	0	0	No Name for File
817cf7c8	20	0	0	0	\$Directory
817cfb98	304	0	0	0	\$Directory
8177be40	16	0	0	0	\$Directory
817dad0c0	2128	68	0	0	\$Mft
817cf008	4	0	0	0	\$Directory
817d0258	8	4	0	0	\$Directory
817763f8	4	0	0	0	\$Directory
...					
8173f058	4	0	0	0	\$Directory
8173e628	32	0	0	0	\$Directory
8173e4c8	32	0	0	0	\$Directory
8173da38	4	0	0	0	\$Directory
817761f8	4	0	0	0	\$Directory
81740530	32	0	0	0	\$Directory
8173d518	4	0	0	0	\$Directory
817d9560	8	0	0	0	\$Directory
8173f868	4	0	0	0	\$Directory
8173fc00	4	0	0	0	\$Directory
81737278	4	0	0	0	\$MftMirr
81737c88	44	0	0	0	\$LogFile
81735fa0	48	0	0	0	\$Mft
81737e88	188	0	0	0	\$BitMap
817380b0	4	0	0	0	\$Mft
817399e0	4	0	0	0	\$Directory
817382b8	4	0	0	0	\$Directory
817388d8	12	0	0	0	No Name for File
81735500	8	0	0	0	\$Directory
81718e38	232	0	0	0	default
81735d40	48	20	0	0	SECURITY
81723008	8632	0	0	0	software
816da3a0	24	44	0	0	SAM
8173dfa0	4	0	0	0	\$Directory
...					
8173ba90	4	0	0	0	\$Directory
8170ee30	4	36	4	0	AppEvent.Evt

816223f8	4	0	0	0	\$Directory
8170ec28	8	28	4	0	SecEvent.Evt
816220a8	4	0	0	0	\$Directory
8170ea20	4	32	4	0	SysEvent.Evt
8170d188	232	0	0	0	NTUSER.DAT
81709f10	8	0	0	0	UsrClass.dat
81708918	232	0	0	0	NTUSER.DAT
81708748	8	0	0	0	UsrClass.dat
816c58f8	12	0	0	0	change.log
815c3880	4	0	0	0	\$Directory
81706aa8	4	0	0	0	SchedLgU.Txt
815ba2d8	4	0	0	0	\$Directory
815aa5f8	8	0	0	0	\$Directory
8166d728	44	0	0	0	Netlogon.log
81701120	8	16	4	0	es.dll
816ff0a8	4	8	4	0	stdole2.tlb
8159a358	4	0	0	0	\$Directory
8159da70	4	0	0	0	\$Directory
8159c158	4	0	0	0	\$Directory
815cb9b0	4	0	0	0	00000001
81779b20	4	0	0	0	\$Directory
8159ac20	4	0	0	0	\$Directory
815683f8	4	0	0	0	\$Directory
81566978	580	0	0	0	NTUSER.DAT
81568460	4	0	0	0	\$Directory
815675d8	68	0	0	0	UsrClass.dat
81567640	4	0	0	0	\$Directory
...					
81515878	4	0	0	0	\$Directory
81516870	8	0	0	0	\$Directory
8150df60	4	0	0	0	\$Directory
...					
816e5300	4	0	0	0	\$Directory
8152afa0	16	212	0	0	msmsgs.exe
8153bbd8	4	32	0	0	stdole32.tlb
8172f950	488	392	0	0	OBJECTS.DATA
8173e9c0	4	0	0	0	\$Directory
814f4538	4	0	0	0	\$Directory
81650790	344	48	0	0	INDEX.BTR
814f55f8	4	0	0	0	\$Directory
...					
814caef8	4	0	0	0	\$Directory
8171cd90	1392	36	0	0	system
815f07f0	4	0	0	0	\$Directory
814a2298	4	0	0	0	\$Directory
81541538	4	0	0	0	\$Directory
81585288	28	0	0	0	\$Directory
8173f708	4	0	0	0	\$Directory
...					
8158cf10	4	0	0	0	\$Directory

!filelock

Article • 10/25/2023

The **!filelock** extension displays a file lock structure.

Syntax

```
dbgcmd  
!filelock FileLockAddress  
!filelock ObjectAddress
```

Parameters

FileLockAddress

Specifies the hexadecimal address of the file lock structure.

ObjectAddress

Specifies the hexadecimal address of a file object that owns the file lock.

DLL

Kdexts.dll

Additional Information

For information about file objects, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

!fileobj

Article • 04/03/2024

The **!fileobj** extension displays detailed information about a FILE_OBJECT structure.

```
dbgcmd
```

```
!fileobj FileObject
```

Parameters

FileObject

Specifies the address of a [FILE_OBJECT](#) structure.

DLL

Kdexts.dll

Additional Information

For information about file objects, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If the FILE_OBJECT structure has an associated cache, **!fileobj** tries to parse and display cache information..

Feedback

Was this page helpful?

 Yes

 No

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

!filetime

Article • 10/25/2023

The **!filetime** extension converts a 64-bit FILETIME structure into a human-readable time.

```
dbgcmd
```

```
!filetime Time
```

Parameters

Time

Specifies a 64-bit FILETIME structure.

DLL

Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !filetime 1c4730984712348
7/26/2004 04:10:18.712 (Pacific Standard Time)
```

!finddata

Article • 10/25/2023

The **!finddata** extension displays the cached data at a given offset within a specified file object.

dbgcmd

```
!finddata FileObject Offset
```

Parameters

FileObject

Specifies the address of the file object.

Offset

Specifies the offset.

DLL

Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the [!icchelp](#) extension.

!findfilelockowner

Article • 10/25/2023

The **!findfilelockowner** extension attempts to find the owner of a file object lock by examining all threads for a thread that is blocked in an **IopSynchronousServiceTail** assert and that has the file object as a parameter.

dbgcmd

```
!findfilelockowner [FileObject]
```

Parameters

FileObject

Specifies the address of a file object. If *FileObject* is omitted, the extension searches for any thread in the current process that is stuck in **IopAcquireFileObjectLock** and retrieves the file object address from the stack trace.

DLL

Kdexts.dll

Additional Information

For information about file objects, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is most useful after a critical section timeout in which the thread that times out was waiting for the file inside **IopAcquireFileObjectLock**. After the offending thread is found, the extension attempts to recover the IRP that was used for the request and to display the driver that was processing the IRP.

The extension takes some time to complete because it walks the stack of all threads in the system until it finds the offending thread. You can stop ` at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

See also

[Displaying a Critical Section](#)

[CriticalSection Time Outs \(user mode\)](#)

!findthreads

Article • 10/25/2023

The !findthreads extension displays summary information about one or more threads on the target system based on supplied search criteria. Thread information will be displayed when the associated stack(s) reference the supplied object. This command can be used only during kernel-mode debugging.

Syntax

```
dbgcmd
```

```
!findthreads [-v][-t <Thread Address>|-i <IRP Address>|-d <Device Address>|  
( -a <Pointer Address> -e <End Address> | -l <Range Length>)]
```

Parameters

-v

Displays verbose information on all criteria matches.

-t ** Thread Address**

The search criteria will be all modules, wait objects, and IRPs for the thread, as well as device objects and modules generated from the attached IRPs. This option will generally provide the broadest search criteria.

-i ** IRP Address**

The search criteria will be all modules and devices for the indicated IRP, as well as any reference to the IRP itself.

-d ** Device Address**

The search criteria will be based from the device object. This will include the module associated with the device object (through the driver object), the device extension, any IRP attached to the device, and similar information for any attached to the device object.

-a ** Pointer Address**

Add a base address to the criteria. If -e or -l is correctly specified, this value will be the base of a memory range. Otherwise it is interpreted as a pointer.

-e ** End Address**

Specifies the end address of the memory range specified with -a.

-I ** Range Length**

Specifies the length of the memory range specified with -a.

DLL

Windows 10 and later

Kdexts.dll

Additional Information

For information about threads in kernel mode, see [Changing Contexts](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is example output using the -v and -t options:

```
dbgcmd
```

```
kd> !findthreads -v -t fffffd001ee29cdc0

Added criterion for THREAD 0xfffffd001ee29cdc0
Added criterion for THREAD STACK 0xfffffd001ee2bac20
ERROR: Object 0xfffffffffffffff0 is not an IRP
ERROR: unable to completely walk thread IRP list.
Added criterion for MODULE kdnic(0xfffff80013120000)

Found 63 threads matching the search criteria

Found 6 criteria matches for THREAD 0xfffffe0016a383740, PROCESS
0xfffffe0016a220200
    Kernel stack location 0xfffffd001f026a0c0 references THREAD
0xfffffd001ee29cdc0
    Kernel stack location 0xfffffd001f026a418 references THREAD
0xfffffd001ee29cdc0
    Kernel stack location 0xfffffd001f026a460 references THREAD
0xfffffd001ee29cdc0
    Kernel stack location 0xfffffd001f026a4d0 references THREAD
0xfffffd001ee29cdc0
    Kernel stack location 0xfffffd001f026a4f0 references THREAD
0xfffffd001ee29cdc0
    Kernel stack location 0xfffffd001f026a670 references THREAD
0xfffffd001ee29cdc0

fffffd001f026a0e0 nt!KiSwapContext+76
fffffd001f026a190 nt!KiSwapThread+1c8
```

```
fffffd001f026a220 nt!KiCommitThreadWait+148
fffffd001f026a2e0 nt!KeWaitForMultipleObjects+21e
fffffd001f026a800 nt!ObWaitForMultipleObjects+2b7
fffffd001f026aa80 nt!NtWaitForMultipleObjects+f6
000000c8d220fa98 nt!KiSystemServiceCopyEnd+13
000000c8d220fa98 ntdll!ZwWaitForMultipleObjects+a
...
```

!for_each_process

Article • 04/03/2024

The `!for_each_process` extension executes the specified debugger command once for each process in the target.

dbgcmd

```
!for_each_process ["CommandString"]
!for_each_process -?
```

Parameters

CommandString

Specifies the debugger commands to be executed for each process.

If *CommandString* includes multiple commands, separate them with semicolons (;) and enclose *CommandString* in quotation marks (""). If *CommandString* is enclosed in quotations marks, the individual commands within *CommandString* cannot contain quotation marks. Within *CommandString*, `@#Process` is replaced by the process address.

-?

Displays help for this extension in the Debugger Command window.

DLL

This extension works only in kernel mode, even though it resides in Ext.dll.

Ext.dll

Additional Information

For general information about processes, see [Threads and Processes](#). For information about manipulating or obtaining information about processes, see [Controlling Processes and Threads](#).

Remarks

If no arguments are supplied, the debugger displays a list of all processes, along with time and priority statistics. This is equivalent to entering [!process @#Process 0](#) as the

CommandString value.

To terminate execution at any point, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!for_each_thread

Article • 04/03/2024

The **!for_each_thread** extension executes the specified debugger command once for each thread in the target.

dbgcmd

```
!for_each_thread ["CommandString"]
!for_each_thread -?
```

Parameters

CommandString

Specifies the debugger commands to be executed for each thread. If *CommandString* includes multiple commands, separate them with semicolons (;) and enclose *CommandString* in quotation marks (""). If *CommandString* is enclosed in quotations marks, the individual commands within *CommandString* cannot contain quotation marks. Within *CommandString*, @#Thread is replaced by the thread address.

-?

Displays help for this extension in the Debugger Command window.

DLL

This extension works only in kernel mode, even though it resides in Ext.dll.

Ext.dll

Additional Information

For more general information about threads, see [Threads and Processes](#). For more information about manipulating or obtaining information about threads, see [Controlling Processes and Threads](#).

Remarks

If no arguments are supplied, the debugger displays a list of all threads, along with thread wait states. This is equivalent to entering **!thread @#Thread 2** as the

CommandString value.

You can terminate execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!fpsearch

Article • 10/25/2023

The **!fpsearch** extension searches the freed special pool for a specified address.

dbgcmd

```
!fpsearch [Address] [Flag]
```

Parameters

Address

Specifies a virtual address.

Flag

If set, the debugger displays the raw content of each page on the free list as it searches the freed special pool.

DLL

Kdexts.dll

Remarks

The display for an address includes the virtual address, the page frame number (PFN), the pool tag, size, whether the data at the address is pageable, the thread ID, and the call stack at the time of deallocation.

If *Address* is set to -1, the debugger displays the entire freed special pool.

If the debugger cannot find the specified address in the freed special pool, it does not display anything. Here is an example of the output from this extension:

dbgcmd

```
kd> !fpsearch -1 1
Searching the free page list (8 entries) for all freed special pool

1EC4000 04000200 e56b6f54 000001f4 0000059c ....Tok.....
1EC4000 00000800 00000000 00000000 00000000 .....
1EC4000 bad0b0b0 82100000 00000000 00000000 .....
1EC4000 72657355 20203233 0000bac5 00000000 User32 .....
```

1EC4000	00028b94	00000000	0000bac9	00000000
1EC4000	00000000	00000000	ffffffffff	7fffffff
1EC4000	8153b1b8	00028aff	00000000	00000000	.S.....
1EC4000	0000001b	00000000	00000012	00000514
26A2000	000a0008	00adecb0	000e000c	00adecba
26A2000	000a0008	00adecc8	000e000c	00adecd2
26A2000	000e000c	00adece0	000e000c	00adecee
26A2000	00120010	00adecfc	000e000c	00aded0e
26A2000	000e000c	00aded1c	000e000c	00aded2a*
26A2000	000e000c	00aded38	000e000c	00aded468.....F..
26A2000	000a0008	00aded54	000e000c	00aded5eT.....^..
26A2000	00120010	00aded6c	000e000c	00aded7e1.....~..
2161000	000a0008	00adeccc	000e000c	00adecd6
2161000	000a0008	00adece4	000e000c	00adecee
2161000	000e000c	00adecfc	000e000c	00aded0a
2161000	00120010	00aded18	000e000c	00aded2a*
2161000	000e000c	00aded38	000e000c	00aded468.....F..
2161000	000e000c	00aded54	000e000c	00aded62T.....b..
2161000	000a0008	00aded70	000e000c	00aded7ap.....z..
2161000	00120010	00aded88	000e000c	00aded9a
 ...					
CEC8000	0311ffa4	03120000	0311c000	00000000
CEC8000	00001e00	00000000	7ff88000	00000000
CEC8000	00000328	00000704	00000000	00000000	(.....
CEC8000	7ffd0f000	00000000	00000000	00000000
CEC8000	e18ba8c0	00000000	00000000	00000000
CEC8000	00000000	00000000	00000000	00000000
CEC8000	00000000	00000000	00000000	00000000
CEC8000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000
CEAD000	00000000	00000000	00000000	00000000

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!frozen

Article • 04/03/2024

The **!frozen** extension displays the state of each processor.

```
dbgcmd
```

```
!frozen
```

DLL

Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
0: kd> !frozen
Processor states:
 0 : Current
 1 : Frozen
```

Feedback

Was this page helpful?

 Yes

 No

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

!fwver

Article • 04/03/2024

The **!fwver** extension displays the Itanium firmware version.

```
dbgcmd
```

```
!fwver
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

DLL

Kdexts.dll

This extension command can only be used with an Itanium target computer.

Additional Information

For more information, consult an Intel architecture manual.

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !fwver
```

```
Firmware Version
```

Sal Revision:	0
SAL_A_VERSION:	0
SAL_B_VERSION:	0
PAL_A_VERSION:	6623
PAL_B_VERSION:	6625
smbiosString:	W460GXBS2.86E.0117A.P08.200107261041

Feedback

Was this page helpful?

 Yes

 No

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

!fxdevice

Article • 04/03/2024

The !fxdevice extension displays summary information about all Power Management Framework (PoFx) registered devices. This command can be used only during kernel-mode debugging.

For more information about PoFX, see [Overview of the Power Management Framework](#).

Syntax

```
dbgcmd
!fxdevice[<FxDevice Address>]
```

Parameters

< FxDevice Address >

Provides the address of the FxDevice to display.

DLL

Kdexts.dll

Remarks

The !fxdevice extension displays the following information when it is present on the target system.

- Non-idle PoFx devices
- Idle D0 PoFx devices
- Idle non-D0 PoFx devices

The following is example output from the !fxdevice extension with a supplied device address.

```
dbgcmd
kd> !fxdevice fffffe0012ccbda60
!fxdevice 0xfffffe0012ccbda60
    DevNode: 0xfffffe0012bbb09f0
```

```

    UniqueId:
    "HDAUDIO\FUNC_01&VEN_10EC&DEV_0662&SUBSYS_103C304A&REV_1001\4&25ff998c&0&000
1"
        InstancePath:
        "HDAUDIO\FUNC_01&VEN_10EC&DEV_0662&SUBSYS_103C304A&REV_1001\4&25ff998c&0&000
1"
            Device Power State: PowerDeviceD0
            PEP Owner: Default PEP
            Acpi Plugin: 0
            Acpi Handle: 0
            Device Status Flags: IdleTimerOn DevicePowerNotRequired_ReceivedFromPEP
            Device Idle Timeout: 0x1869ffffe7960
            Device Power On: No Activity
            Device Power Off: No Activity
            Device Unregister: No Activity
            Component Count: 1
                Component 0: F0/F0 - IDLE (RefCount = 0)
                Pep Component: 0xfffffe0012cf1800
                    Active: 0 Latency: 0 Residency: 0 Wake: 0 Dx IRP: 0 WW
            IRP: 0
                Component Idle State Change: No Activity
                Component Activation: No Activity
                Component Active: No Activity

```

The following is the default output from the !fxdevice extension.

```

dbgcmd

kd> !fxdevice
*****
*** Dumping non-idle PoFx devices ****
!fxdevice 0xfffffe0012bbd08d0
    DevNode: 0xfffffe0012b3f87b0
    UniqueId: "\_SB.PCI0"
    InstancePath: "ACPI\PNP0A08\2&daba3ff&1"
    Device Power State: PowerDeviceD0
    Component Count: 1
        Component 0: F0/F1 - ACTIVE (RefCount = 28)

!fxdevice 0xfffffe0012c587940
    DevNode: 0xfffffe0012b3f9d30
    UniqueId: "\_SB.PCI0.PEGL"
    InstancePath:
    "PCI\VEN_8086&DEV_D138&SUBSYS_304A103C&REV_11\3&33fd14ca&0&18"
        Device Power State: PowerDeviceD0
        Component Count: 1
            Component 0: F0/F1 - ACTIVE (RefCount = 1)

...

```

```
*****
Dumping idle D0 PoFx devices
*****
****

!fxdevice 0xfffffe0012c5838c0
    DevNode: 0xfffffe0012bbdf30
    UniqueId: "\_SB.PCI0.PCX1"
    InstancePath:
"PCI\VEN_8086&DEV_3B42&SUBSYS_304A103C&REV_05\3&33fd14ca&0&E0"
        Device Power State: PowerDeviceD0
        Component Count: 1
            Component 0: F1/F1 - IDLE (RefCount = 0)

!fxdevice 0xfffffe0012c581ac0
    DevNode: 0xfffffe0012bbdfa50
    UniqueId: "\_SB.PCI0.PCX5"
    InstancePath:
"PCI\VEN_8086&DEV_3B4A&SUBSYS_304A103C&REV_05\3&33fd14ca&0&E4"
        Device Power State: PowerDeviceD0
        Component Count: 1
            Component 0: F1/F1 - IDLE (RefCount = 0)

...

```

Feedback

Was this page helpful?



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

!gbl

Article • 04/03/2024

The **!gbl** extension displays header information from the ACPI BIOS Root System Description (RSDT) table of the target computer.

```
dbgcmd
```

```
!gbl [-v]
```

Parameters

-v

Verbose. Displays detailed information about the table.

DLL

Kdexts.dll

Additional Information

For information about the ACPI and ACPI tables, see [Other ACPI Debugging Extensions](#) and the [ACPI Specification](#) Web site. Also see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Feedback

Was this page helpful?

 Yes

 No

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

!gentable

Article • 10/25/2023

The **!gentable** extension displays an RTL_GENERIC_TABLE.

Syntax

dbgcmd

```
!gentable Address[Flag]
```

Parameters

Address

Specifies the address of the RTL_GENERIC_TABLE.

Flag

Specifies the table source. If *Flag* is 1, the AVL table is used. If *Flag* is 0 or omitted, the non-AVL table is used.

DLL

Kdexts.dll

!hidppd

Article • 04/03/2024

The **!hidppd** extension command displays the contents of the HIDP_PREPARSED_DATA structure.

dbgcmd

!hidppd Address

Parameters

Address

Specifies the hexadecimal address of the HIDP_PREPARSED_DATA structure.

DLL

Kdexts.dll

Additional Information

For information about human input devices (HID), see the Windows Driver Kit (WDK) documentation.

Feedback

Was this page helpful?

 Yes

 No

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

!ib, !id, !iw

Article • 10/25/2023

The **!ib**, **!id**, and **!iw** extension commands are obsolete. Use the [**ib, id, iw \(Input from Port\)**](#) commands instead.

!icpleak

Article • 04/03/2024

The `!icpleak` extension examines all I/O completion objects in the system for the object with the largest number of queued entries.

dbgcmd

```
!icpleak [HandleFlag]
```

Parameters

HandleFlag

If this flag is set, the display also includes all processes that have a handle to the object with the largest number of queued entries.

DLL

Kdexts.dll

Additional Information

For information about I/O completion ports, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is useful when there is a leak in the I/O completion pool. I/O completion pool leaks can occur when a process is allocating I/O completion packets by calling [PostQueuedCompletionStatus](#), but is not calling [GetQueuedCompletionStatus](#) to free them, or when a process is queuing completion entries to a port, but there is no thread retrieving the entries. To detect a leak run the [!poolused](#) extension and check the value of ICP pool tag. If pool use with the ICP tag is significant, a leak might have occurred.

This extension works only if the system maintains type lists. If the *HandleFlag* is set and the system has many processes, this extension will take a long time to run.

You can stop at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!idt

Article • 04/03/2024

The **!idt** extension displays the interrupt service routines (ISRs) for a specified interrupt dispatch table (IDT).

dbgcmd

```
!idt IDT  
!idt [-a]  
!idt -?
```

Parameters

IDT

Specifies the IDT to display.

-a

When *IDT* is not specified, the debugger displays the IDTs of all processors on the target computer in an abbreviated format. If **-a** is specified, the ISRs for each IDT are also displayed.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

This extension command can only be used with an x64-based or x86-based target computer.

Additional Information

For information about ISRs and IDTs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
0: kd> !idt

Dumping IDT:

37:806ba78c hal!PicSpuriousService37
3d:806bbc90 hal!HalpApcInterrupt
41:806bbb04 hal!HalpDispatchInterrupt
50:806ba864 hal!HalpApicRebootService
63:8641376c VIDEOOPRT!pVideoPortInterrupt (KINTERRUPT 86413730)
73:862aa044 portcls!CInterruptSyncServiceRoutine (KINTERRUPT 862aa008)
82:86594314 atapi!IdePortInterrupt (KINTERRUPT 865942d8)
83:86591bec SCSIPORT!ScsiPortInterrupt (KINTERRUPT 86591bb0)
92:862b53dc serial!SerialCIrSw (KINTERRUPT 862b53a0)
93:86435844 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 86435808)
a3:863b366c i8042prt!I8042MouseInterruptService (KINTERRUPT 863b3630)
a4:8636bbec USBPORT!USBPORT_InterruptService (KINTERRUPT 8636bbb0)
b1:86585bec ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 86585bb0)
b2:863c0524 serial!SerialCIrSw (KINTERRUPT 863c04e8)
b4:86391a54 NDIS!ndisMIsr (KINTERRUPT 86391a18)
          USBPORT!USBPORT_InterruptService (KINTERRUPT 863ae890)
c1:806ba9d0 hal!HalpBroadcastCallService
d1:806b9dd4 hal!HalpClockInterrupt
e1:806baf30 hal!HalpIpiHandler
e3:806bacaa hal!HalpLocalApicErrorService
fd:806bb460 hal!HalpProfileInterrupt
```

Feedback

Was this page helpful?

 Yes

 No

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

!ih

Article • 04/03/2024

The **!ih** extension displays the interrupt history record for the specified processor.

```
dbgcmd
```

```
  !ih Processor
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor

Specifies a processor. If *Processor* is omitted, the current processor is used.

DLL

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

This extension displays the interrupt history record without referencing the program counter symbols. To display the interrupt history record using the program counter symbols, use the [!ihs](#) extension. To enable the interrupt history record, add **/configflag=32** to the boot entry options.

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !ih
Total # of interruptions = 2093185
Vector          IIP           IPSR           ExtraField
VHPT FAULT    e0000000830d3190    1010092a6018  IFA=      6fc00a0200c
          VHPT FAULT    e0000000830d33d0    1010092a6018  IFA=
1fffffe00001de2d0
          VHPT FAULT    e0000000830d33d0    1010092a6018  IFA= 1fffffe01befff338
          VHPT FAULT    e0000000830d3190    1010092a6018  IFA=
```

6fc00a0200c				
VHPT FAULT	e0000000830d33d0	1010092a6018	IFA=	
1fffffe00001d9188				
VHPT FAULT	e0000000830d3880	1010092a6018	IFA=	
1fffffe01befff250				
VHPT FAULT	e0000000830d3fb0	1010092a6018	IFA=	
e0000165f82dc1c0				
VHPT FAULT	e000000083063390	1010092a6018	IFA= e0000000ffffe0018	
THREAD SWITCH	e000000085896040	e00000008588c040	OSP=	
e0000165f82dbd40				
VHPT FAULT	e00000008329b130	1210092a6018	IFA=	
e0000165f7edaf30				
VHPT FAULT	e0000165f7eda640	1210092a6018	IFA=	
e0000165ffff968a9				
PROCESS SWITCH	e0000000818bbe10	e000000085896040	OSP=	
e0000165f8281de0				
VHPT FAULT	e00000008307cf0	1010092a2018	IFA=	
e00000008189fe50				
EXTERNAL INTERRUPT	e0000000830623b0	1010092a6018	IVR=	
d0				
VHPT FAULT	e00000008314e310	1010092a2018	IFA=	
e0000165f88203f0				
VHPT FAULT	e000000083580760	1010092a2018	IFA=	
e0000000f00ff3fd				
PROCESS SWITCH	e00000008558c990	e0000000818bbe10	OSP=	
e00000008189fe20				
VHPT FAULT	e00000008307cf0	1010092a2018	IFA=	
e0000165f02433f0				
VHPT FAULT	77cfbda0	1013082a6018	IFA=	
77cfbda0				
VHPT FAULT	77cfbdb0	1213082a6018	IFA= 6fbfee0ff98	
DATA ACCESS BIT	77b8e150	1213082a6018	IFA=	
77c16ab8				
VHPT FAULT	77ed5d60	1013082a6018	IFA=	
6fbffffa6048				
DATA ACCESS BIT	77ed5d60	1213082a6018	IFA=	
77c229c0				
DATA ACCESS BIT	77b8e1b0	1013082a6018	IFA= 77c1c320	
USER SYSCALL	77cafa40	10082a6018	Num=	
42				
VHPT FAULT	e00000008344dc20	1010092a6018	IFA= e000010600703784	
...				

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!ihs

Article • 04/03/2024

The **!ihs** extension displays the interrupt history record for the specified processor, using program counter symbols.

```
dbgcmd
```

```
  !ihs Processor
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Processor

Specifies a processor. If *Processor* is omitted, the current processor is used.

DLL

Kdexts.dll

This extension command can only be used with an Itanium target computer.

Remarks

To display the interrupt history record without using program counter symbols, use the **!ih** extension. To enable the interrupt history record, add **/configflag=32** to the boot entry options.

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !ihs
Total # of interruptions = 2093185
Vector          IIP          IPSR          ExtraField
IIP Symbol
    VHPT FAULT  e0000000830d3190      1010092a6018  IFA=
6fc00a0200c nt!MiAgeAndEstimateAvailableInWorkingSet+0x70
    VHPT FAULT  e0000000830d33d0      1010092a6018  IFA=
1fffffe00001de2d0 nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
```

```

        VHPT FAULT e0000000830d33d0      1010092a6018 IFA=
1fffffe01befff338 nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
        VHPT FAULT e0000000830d3190      1010092a6018 IFA=
6fc00a0200c nt!MiAgeAndEstimateAvailableInWorkingSet+0x70
        VHPT FAULT e0000000830d33d0      1010092a6018 IFA=
1fffffe00001d9188 nt!MiAgeAndEstimateAvailableInWorkingSet+0x2b0
        VHPT FAULT e0000000830d3880      1010092a6018 IFA=
1fffffe01befff250 nt!MiAgeAndEstimateAvailableInWorkingSet+0x760
        VHPT FAULT e0000000830d3fb0      1010092a6018 IFA=
e0000165f82dc1c0 nt!MiCheckAndSetSystemTrimCriteria+0x190
        VHPT FAULT e000000083063390      1010092a6018 IFA=
e0000000ffffe0018 nt!KeQuerySystemTime+0x30
        THREAD SWITCH e000000085896040 e00000008588c040 OSP=
e0000165f82dbd40
        VHPT FAULT e00000008329b130      1210092a6018 IFA=
e0000165f7edaf30 nt!IopProcessWorkItem+0x30
        VHPT FAULT e0000165f7eda640      1210092a6018 IFA= e0000165fff968a9
netbios!RunTimerForLana+0x60
        PROCESS SWITCH e0000000818bbe10 e000000085896040 OSP=
e0000165f8281de0
        VHPT FAULT e00000008307cf0      1010092a2018 IFA=
e00000008189fe50 nt!SwapFromIdle+0x1e0
        EXTERNAL INTERRUPT e0000000830623b0      1010092a6018 IVR=
d0 nt!Kill_TopOfIdleLoop
        VHPT FAULT e00000008314e310      1010092a2018 IFA=
e0000165f88203f0 nt!KdReceivePacket+0x10
        VHPT FAULT e000000083580760      1010092a2018 IFA=
e0000000f00ff3fd hal!READ_PORT_UCHAR+0x80
        PROCESS SWITCH e00000008558c990 e0000000818bbe10 OSP=
e00000008189fe20
        VHPT FAULT e00000008307cf0      1010092a2018 IFA=
e0000165f02433f0 nt!SwapFromIdle+0x1e0
        VHPT FAULT 77cfbda0      1013082a6018 IFA=
77cfbda0 0x0000000077cfbda0
        VHPT FAULT 77cfbdb0      1213082a6018 IFA=
6fbfee0ff98 0x0000000077cfbdb0
        DATA ACCESS BIT 77b8e150      1213082a6018 IFA= 77c16ab8
0x0000000077b8e150
        VHPT FAULT 77ed5d60      1013082a6018 IFA= 6fbffffa6048
0x0000000077ed5d60
        DATA ACCESS BIT 77ed5d60      1213082a6018 IFA=
77c229c0 0x0000000077ed5d60
        DATA ACCESS BIT 77b8e1b0      1013082a6018 IFA= 77c1c320
0x0000000077b8e1b0
        USER SYSCALL 77cafa40      10082a6018 Num=
42 0x0000000077cafa40
        VHPT FAULT e00000008344dc20      1010092a6018 IFA= e000010600703784
nt!ExpLookupHandleTableEntry+0x20
...

```

Feedback

Was this page helpful?

 Yes

 No

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

!ioresdes

Article • 10/25/2023

The **!ioresdes** extension displays the IO_RESOURCE_DESCRIPTOR structure at the specified address.

```
dbgcmd
```

```
!ioresdes Address
```

Parameters

Address

Specifies the hexadecimal address of the IO_RESOURCE_DESCRIPTOR structure.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about the IO_RESOURCE_DESCRIPTOR structure, see the Windows Driver Kit (WDK) documentation.

!ioctldecode

Article • 10/25/2023

The **!ioctldecode** extension displays the *Device Type*, *Required Access*, *Function Code* and *Transfer Type* as specified by the given IOCTL code. For more information about IOCTL control codes, see [Defining I/O Control Codes](#).

```
dbgcmd
```

```
!ioctldecode IoctlCode
```

Parameters

IoctlCode

Specifies the hexadecimal IOCTL Code. The [!irp](#) command displays the IOCTL code in its output.

DLL

Kdexts.dll

Additional Information

To see information on the IOCTL, we first locate an IRP of interest. You can use the [!irpfind](#) command to locate an irp of interest.

Use the [!irp](#) command to display information about the irp.

```
dbgcmd
```

```
0: kd> !irp fffffd581a6c6cd30
Irp is active with 6 stacks 6 is current (= 0xfffffd581a6c6cf68)
No Mdl: No System Buffer: Thread 00000000: Irp stack trace.
    cmd   flg cl Device   File      Completion-Context
[N/A(0), N/A(0)]
        0  0 00000000 00000000 00000000-00000000
```

Args: 00000000 00000000

```
00000000 00000000
[N/A(0), N/A(0)]
        0  0 00000000 00000000 00000000-00000000
```

Args: 00000000 00000000

```
00000000 00000000
[N/A(0), N/A(0)]
    0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000

00000000 00000000
[N/A(0), N/A(0)]
    0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000

00000000 00000000
[N/A(0), N/A(0)]
    0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000

00000000 00000000
>[IRP_MJ_INTERNAL_DEVICE_CONTROL(f), N/A(0)]
    0 e1 fffffd581a5fdb050 00000000 fffff806d2412cf0-fffffd581a5cce050
Success Error Cancel pending
                \Driver\usbehci          (IoUnloadSafeCompletion)
                                                Args: fffffd581a6c61a50
00000000 0x220003 00000000
```

The third argument displayed, in this case *0x220003*, is the IOCTL code. Use the IOCTL code to display information about the IOCTL, in this case

[IOCTL_INTERNAL_USB_SUBMIT_URB](#).

```
dbgcmd

0: kd> !ioctldecode 0x220003

IOCTL_INTERNAL_USB_SUBMIT_URB

Device Type      : 0x22 (FILE_DEVICE_WINLOAD) (FILE_DEVICE_USER_MODE_BUS)
(FILE_DEVICE_USB) (FILE_DEVICE_UNKNOWN)
Method          : 0x3 METHOD_NEITHER
Access          : FILE_ANY_ACCESS
Function        : 0x0
```

If you provide an IOCTL code that is not available, you will see this type of output.

```
dbgcmd

0: kd> !ioctldecode 0x1280ce

Unknown IOCTL  : 0x1280ce

Device Type    : 0x12 (FILE_DEVICE_NETWORK)
Method         : 0x2 METHOD_OUT_DIRECT
```

```
Access      : FILE_WRITE_ACCESS
Function    : 0x33
```

Although the IOCTL is not identified, information about the IOCTL fields are displayed.

Note that only a subset of publicly defined IOCTLs are able to be identified by the **!ioctldecode** command.

For more information about IOCTLs see [Introduction to I/O Control Codes](#).

For more general information about IRPs and IOCTLs, refer to *Windows Internals* by Mark E. Russinovich, David A. Solomon and Alex Ionescu.

!ioreslist

Article • 10/25/2023

The **!ioreslist** extension displays an IO_RESOURCE_REQUIREMENTS_LIST structure.

```
dbgcmd
```

```
!ioreslist Address
```

Parameters

Address

Specifies the hexadecimal address of the IO_RESOURCE_REQUIREMENTS_LIST structure.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about the IO_RESOURCE_REQUIREMENTS_LIST structure, see the Windows Driver Kit (WDK) documentation.

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !ioreslist 0xe122b768
```

```
IoResList at 0xe122b768 : Interface 0x5 Bus 0 Slot 0xe
Alternative 0 (Version 1.1)
Preferred Descriptor 0 - Port (0x1) Device Exclusive (0x1)
Flags (0x01) - PORT_IO
0x0000100 byte range with alignment 0x0000100
1000 - 0x10ff
Alternative Descriptor 1 - Port (0x1) Device Exclusive (0x1)
Flags (0x01) - PORT_IO
0x0000100 byte range with alignment 0x0000100
0 - 0xffffffff
```

```

Descriptor 2 - DevicePrivate (0x81) Device Exclusive (0x1)
Flags (0000) -
Data:           : 0x1 0x0 0x0
Preferred Descriptor 3 - Memory (0x3) Device Exclusive (0x1)
Flags (0000) - READ_WRITE
0x001000 byte range with alignment 0x001000
40080000 - 0x40080fff
Alternative Descriptor 4 - Memory (0x3) Device Exclusive (0x1)
Flags (0000) - READ_WRITE
0x001000 byte range with alignment 0x001000
0 - 0xffffffff
Descriptor 5 - DevicePrivate (0x81) Device Exclusive (0x1)
Flags (0000) -
Data:           : 0x1 0x1 0x0
Descriptor 6 - Interrupt (0x2) Shared (0x3)
Flags (0000) - LEVEL_SENSITIVE
0xb - 0xb

```

The IO_RESOURCE_REQUIREMENTS_LIST contains information about:

- Resource types

There are four types of resources: I/O, Memory, IRQ, DMA.

- Descriptors

Each preferred setting has a "Preferred" descriptor and a number of "Alternative" descriptors.

This resource list contains the following requests:

- I/O Ranges

Prefers a range of 0x1000 to 0x10FF inclusive, but can use any 0x100 range between 0 and 0xFFFFFFFF, provided it is 0x100-aligned. (For example, 0x1100 to 0x11FF is acceptable.)

- Memory

Prefers a range of 0x40080000 to 0x40080FFF, but can use any range that is of size 0x1000, is 0x1000-aligned, and is located between 0 and 0xFFFFFFFF.

- IRQ

Must use IRQ 0xB.

Interrupts and DMA channels are represented as ranges with the same beginning and end.

!iovirp

Article • 04/03/2024

The `!iovirp` extension displays detailed information for a specified I/O Verifier IRP.

```
dbgcmd
```

```
!iovirp [IRP]
```

Parameters

IRP

Specifies the address of an IRP tracked by the Driver Verifier. If *IRP* is 0 or is omitted, the summary information for each outstanding IRP is displayed.

DLL

Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !iovirp 947cef68
IovPacket      84509af0
TrackedIrp     947cef68
HeaderLock     84509d61
LockIrql       0
ReferenceCount 1
PointerCount   1
HeaderFlags    00000000
ChainHead      84509af0
Flags          00200009
DepartureIrql  0
ArrivalIrql    0
StackCount     1
QuotaCharge    00000000
QuotaProcess   0
RealIrpCompletionRoutine 0
RealIrpControl 0
RealIrpContext 0
TopStackLocation 2
```

```
PriorityBoost          0
LastLocation          0
RefTrackingCount      0
SystemDestVA          0
VerifierSettings      84509d08
pIovSessionData       84509380
Allocation Stack:
  nt!IovAllocateIrp+1a  (817df356)
  nt!IopXxxControlFile+40c  (8162de20)
  nt!NtDeviceIoControlFile+2a  (81633090)
  nt!KiFastCallEntry+164  (81513c64)
  nt!EtwpFlushBuffer+10f  (817606d7)
  nt!EtwpFlushBuffersWithMarker+bd  (817608cb)
  nt!EtwpFlushActiveBuffers+2b4  (81760bc2)
  nt!EtwpLogger+213  (8176036f)
```

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!ipi

Article • 04/03/2024

The **!ipi** extension displays the interprocessor interrupt (IPI) state for a specified processor.

```
dbgcmd
```

```
    !ipi [Processor]
```

Parameters

Processor

Specifies a processor. If *Processor* is omitted, the IPI state for every processor is displayed.

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For information about IPIs, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
0: kd> !ipi
IPI State for Processor 0
Worker Routine: nt!KiFlushTargetMultipleTb [Stale]
Parameter[0]: 0
Parameter[1]: 3
Parameter[2]: F7C98770
Ipi Trap Frame: F7CCCCDC [.trap F7CCCCDC]
Signal Done: 0
```

```
IPI Frozen:      24 [FreezeActive] [Owner]
Request Summary: 0
Target Set:      0
Packet Barrier:  0

IPI State for Processor 1
Worker Routine:  nt!KiFlushTargetMultipleTb [Stale]
Parameter[0]:    1
Parameter[1]:    3
Parameter[2]:    F7CDCD28
Ipi Trap Frame: F7C8CCC4 [.trap F7C8CCC4]
Signal Done:     0
IPI Frozen:      2 [Frozen]
Request Summary: 0
Target Set:      0
Packet Barrier:  0
```

Feedback

Was this page helpful?

 Yes

 No

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

!irp

Article • 10/25/2023

The **!irp** extension displays information about an I/O request packet (IRP).

dbgcmd

!irp Address [Detail]

Parameters

Address

Specifies the hexadecimal address of the IRP.

Detail

If this parameter is included with any value, such as 1, the output includes the status of the IRP, the address of its memory descriptor list (MDL), its owning thread, and stack information for all of its I/O stacks, and information about each stack location for the IRP, including hexadecimal versions of the major function code and the minor function code. If this parameter omitted, the output includes only a summary of the information.

DLL

Windows XP and later

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) and [Debugging Interrupt Storms](#) for applications of this extension command. For information about IRPs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. For further information on the major and minor function codes, see the Windows Driver Kit (WDK) documentation.

This topic describes the IRP structure, [IRP](#).

For detailed information on decoding the IRP structure including the returned Args, see the following resources.

- Windows Internals by Mark E. Russinovich, David A. Solomon and Alex Ionescu

- Developing Drivers with the Windows Driver Foundation Guy Smith and Penny Orwick

Remarks

The output also indicates under what conditions the completion routine for each stack location will be called once the IRP has completed and the stack location is processed. There are three possibilities:

Success

Indicates that the completion routine will be called when the IRP is completed with a success code.

Error

Indicates that the completion routine will be called when the IRP is completed with an error code.

Cancel

Indicates that the completion routine will be called if an attempt was made to cancel the IRP.

Any combination of these three may appear, and if any of the conditions shown are satisfied, the completion routine will be called. The appropriate values are listed at the end of the first row of information about each stack location, immediately after the **Completion-Context** entry.

Here is an example of the output from this extension for Windows 10:

```
dbgcmd

0: kd> !irp ac598dc8
Irp is active with 2 stacks 1 is current (= 0xac598e38)
No Mdl: No System Buffer: Thread 8d1c7bc0: Irp stack trace.
    cmd   flg  cl Device   File      Completion-Context
>[IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
    1 e1 8a6434d8 ac598d40 853220cb-a89682d8 Success Error Cancel
pending
    \FileSystem\Npfs fltmgr!FltpPassThroughCompletion
        Args: 00000000 00000000 00110008 00000000
[IRP_MJ_FILE_SYSTEM_CONTROL(d), N/A(0)]
    1 0 8a799710 ac598d40 00000000-00000000
    \FileSystem\FltMgr
        Args: 00000000 00000000 0x00110008 00000000
```

Starting with Windows 10 the IRP major and minor code text is displayed, for example, "IRP_MJ_FILE_SYSTEM_CONTROL" The code value is also shown in hex, in this example "

(d)".

The third argument displayed in the output, is the IOCTL code. Use the [!ioctldecode](#) command to display information about the IOCTL.

Here is an example of the output from this extension from Windows Vista.

```
dbgcmd

0: kd> !irp 0x831f4a00
Irp is active with 8 stacks 5 is current (= 0x831f4b00)
Mdl = 82b020d8 Thread 8c622118: Irp stack trace.
    cmd   flg cl Device     File      Completion-Context
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000 00000000 00000000
[ 0, 0] 0 0 00000000 00000000 00000000-00000000

                                                Args: 00000000 00000000 00000000 00000000
>[ 3,34] 40 e1 828517a8 00000000 842511e0-00000000 Success Error Cancel
pending
          \Driver\disk      partmgr!PmReadWriteCompletion
Args: 00007000 00000000 fe084e00 00000004
[ 3, 0] 40 e0 82851450 00000000 842414d4-82956350 Success Error Cancel
\Driver\PartMgr  volmgr!VmpReadWriteCompletionRoutine
          Args: 129131bb 000000de fe084e00 00000004
[ 3, 0] 0 e0 82956298 00000000 847eed0-829e2ba8 Success Error Cancel
\Driver\volmgr  Ntfs!NtfsMasterIrpSyncCompletionRoutine
          Args: 00007000 00000000 1bdae400 00000000
[ 3, 0] 0 0 82ac2020 8e879410 00000000-00000000
          \FileSystem\Ntfs
          Args: 00007000 00000000 00018400 00000000
```

Note that the completion routine next to the driver name is set on that stack location, and it was set by the driver in the line below. In the preceding example, **Ntfs!NtfsMasterIrpSyncCompletionRoutine** was set by **\FileSystem\Ntfs**. The **Completion-Context** entry above **Ntfs!NtfsMasterIrpSyncCompletionRoutine**, **847eed0-829e2ba8**, indicates the address of the completion routine, as well as the context that will be passed to **Ntfs!NtfsMasterIrpSyncCompletionRoutine**. From this we can see that the address of **Ntfs!NtfsMasterIrpSyncCompletionRoutine** is **847eed0**, and the context that will be passed to this routine when it is called is **829e2ba8**.

IRP major function codes

The following information is included to help you interpret the output from this extension command.

The IRP major function codes are as follows:

Major Function Code	Hexadecimal Code
IRP_MJ_CREATE	0x00
IRP_MJ_CREATE_NAMED_PIPE	0x01
IRP_MJ_CLOSE	0x02
IRP_MJ_READ	0x03
IRP_MJ_WRITE	0x04
IRP_MJ_QUERY_INFORMATION	0x05
IRP_MJ_SET_INFORMATION	0x06
IRP_MJ_QUERY_EA	0x07
IRP_MJ_SET_EA	0x08
IRP_MJ_FLUSH_BUFFERS	0x09
IRP_MJ_QUERY_VOLUME_INFORMATION	0x0A
IRP_MJ_SET_VOLUME_INFORMATION	0x0B
IRP_MJ_DIRECTORY_CONTROL	0x0C
IRP_MJ_FILE_SYSTEM_CONTROL	0x0D
IRP_MJ_DEVICE_CONTROL	0x0E
IRP_MJ_INTERNAL_DEVICE_CONTROL	0x0F
IRP_MJ_SCSI	
IRP_MJ_SHUTDOWN	0x10
IRP_MJ_LOCK_CONTROL	0x11
IRP_MJ_CLEANUP	0x12
IRP_MJ_CREATE_MAILSLOT	0x13
IRP_MJ_QUERY_SECURITY	0x14
IRP_MJ_SET_SECURITY	0x15
IRP_MJ_POWER	0x16

Major Function Code	Hexadecimal Code
IRP_MJ_SYSTEM_CONTROL	0x17
IRP_MJ_DEVICE_CHANGE	0x18
IRP_MJ_QUERY_QUOTA	0x19
IRP_MJ_SET_QUOTA	0x1A
IRP_MJ_PNP IRP_MJ_MAXIMUM_FUNCTION	0x1B

The Plug and Play minor function codes are as follows:

Minor Function Code	Hexadecimal Code
IRP_MN_START_DEVICE	0x00
IRP_MN_QUERY_REMOVE_DEVICE	0x01
IRP_MN_REMOVE_DEVICE	0x02
IRP_MN_CANCEL_REMOVE_DEVICE	0x03
IRP_MN_STOP_DEVICE	0x04
IRP_MN_QUERY_STOP_DEVICE	0x05
IRP_MN_CANCEL_STOP_DEVICE	0x06
IRP_MN_QUERY_DEVICE_RELATIONS	0x07
IRP_MN_QUERY_INTERFACE	0x08
IRP_MN_QUERY_CAPABILITIES	0x09
IRP_MN_QUERY_RESOURCES	0x0A
IRP_MN_QUERY_RESOURCE_REQUIREMENTS	0x0B
IRP_MN_QUERY_DEVICE_TEXT	0x0C
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	0x0D
IRP_MN_READ_CONFIG	0x0F
IRP_MN_WRITE_CONFIG	0x10
IRP_MN_EJECT	0x11
IRP_MN_SET_LOCK	0x12
IRP_MN_QUERY_ID	0x13

Minor Function Code	Hexadecimal Code
IRP_MN_QUERY_PNP_DEVICE_STATE	0x14
IRP_MN_QUERY_BUS_INFORMATION	0x15
IRP_MN_DEVICE_USAGE_NOTIFICATION	0x16
IRP_MN_SURPRISE_REMOVAL	0x17
IRP_MN_QUERY_LEGACY_BUS_INFORMATION	0x18

The WMI minor function codes are as follows:

Minor Function Code	Hexadecimal Code
IRP_MN_QUERY_ALL_DATA	0x00
IRP_MN_QUERY_SINGLE_INSTANCE	0x01
IRP_MN_CHANGE_SINGLE_INSTANCE	0x02
IRP_MN_CHANGE_SINGLE_ITEM	0x03
IRP_MN_ENABLE_EVENTS	0x04
IRP_MN_DISABLE_EVENTS	0x05
IRP_MN_ENABLE_COLLECTION	0x06
IRP_MN_DISABLE_COLLECTION	0x07
IRP_MN_REGINFO	0x08
IRP_MN_EXECUTE_METHOD	0x09

The power management minor function codes are as follows:

Minor Function Code	Hexadecimal Code
IRP_MN_WAIT_WAKE	0x00
IRP_MN_POWER_SEQUENCE	0x01
IRP_MN_SET_POWER	0x02
IRP_MN_QUERY_POWER	0x03

The SCSI minor function codes are as follows:

Minor Function Code	Hexadecimal Code
IRP_MN_SCSI_CLASS	0x01

See also

[IRP](#)

[!irpfind](#)

[!ioctldecode](#)

!irpfind

Article • 10/25/2023

The **!irpfind** extension displays information about all I/O request packets (IRP) currently allocated in the target system, or about those IRPs matching the specified search criteria.

Syntax

```
dbgcmd
```

```
!irpfind [-v][PoolType[RestartAddress[CriteriaData]]]
```

Parameters

-v

Displays verbose information.

PoolType

Specifies the type of pool to be searched. The following values are permitted:

0

Specifies nonpaged memory pool. This is the default.

1

Specifies paged memory pool.

2

Specifies the special pool.

4

Specifies the session pool.

RestartAddress

Specifies the hexadecimal address at which to begin the search. This is useful if the previous search was terminated prematurely. The default is zero.

Criteria

Specifies the criteria for the search. Only those IRPs that satisfy the given match will be displayed.

Criteria	Match
arg	Finds all IRPs with a stack location where one of the arguments equals <i>Data</i> .
device	Finds all IRPs with a stack location where DeviceObject equals <i>Data</i> .
fileobject	Finds all IRPs whose Irp.Tail.Overlay.OriginalFileObject equals <i>Data</i> .
mdlprocess	Finds all IRPs whose Irp.MdlAddress.Process equals <i>Data</i> .
thread	Finds all IRPs whose Irp.Tail.Overlay.Thread equals <i>Data</i> .
userevent	Finds all IRPs whose Irp.UserEvent equals <i>Data</i> .

Data

Specifies the data to be matched in the search.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about IRPs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This example finds the IRP in the nonpaged pool that is going to set user event FF9E4F48 upon completion:

```
dbgcmd
kd> !irpfind 0 0 userevent ff9e4f48
```

The following example produces a full listing of all IRPs in the nonpaged pool:

```
dbgcmd
```

```
kd> !irpfind
Searching NonPaged pool (8090c000 : 8131e000) for Tag: Irp
8097c008 Thread 8094d900 current stack belongs to \Driver\symc810
8097dec8 Thread 8094dda0 current stack belongs to \FileSystem\Ntfs
809861a8 Thread 8094dda0 current stack belongs to \Driver\symc810
809864e8 Thread 80951ba0 current stack belongs to \Driver\Mouclass
80986608 Thread 80951ba0 current stack belongs to \Driver\Kbdclass
80986728 Thread 8094dda0 current stack belongs to \Driver\symc810
```

!irpzone

Article • 10/25/2023

The **!irpzone** extension command is obsolete. Use [**!irpfind**](#) instead.

!irql

Article • 10/25/2023

The **!irql** extension displays the interrupt request level (IRQL) of a processor on the target computer before the debugger break.

dbgcmd

!irql [Processor]

Parameters

Processor

Specifies the processor. Enter the processor number. If this parameter is omitted, the debugger displays the IRQL of the current processor.

DLL

The **!irql** extension is only available in Windows Server 2003 and later versions of Windows.

Windows 2000	Unavailable
Windows XP	Unavailable
Windows Server 2003 and later	Kdexts.dll

Additional Information

For information about IRQLs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

When the target computer breaks into the debugger, the IRQL changes, but the IRQL that was effective just before the debugger break is saved. The **!irql** extension displays the saved IRQL.

Similarly, when a bug check occurs and a crash dump file is created, the IRQL saved in the crash dump file is the one immediately prior to the bug check, not the IRQL at which

the [KeBugCheckEx](#) routine was executed.

In both cases, the current IRQL is raised to DISPATCH_LEVEL, except on x86 architectures. Thus, if more than one such event occurs, the IRQL displayed will also be DISPATCH_LEVEL, making it useless for debugging purposes.

The [!pcr](#) extension displays the current IRQL on all versions of Windows, but the current IRQL is usually not useful. The IRQL that existed just before the bug check or debugger connection is more interesting, and this is displayed only with [!irql](#).

If you supply an invalid processor number, or there has been kernel corruption, the debugger displays a message "Cannot get PRCB address".

Here is an example of the output from this extension from a dual-processor x86 computer:

```
dbgcmd

kd> !irql 0
Debugger saved IRQL for processor 0x0 -- 28 (CLOCK2_LEVEL)

kd> !irql 1
Debugger saved IRQL for processor 0x1 -- 0 (LOW_LEVEL)
```

If the debugger is in verbose mode, a description of the IRQL itself is included.

The meaning of the IRQL number often depends on the processor. Here is an example from an x64 processor. Note that the IRQL number is the same as in the previous example, but the IRQL meaning is different:

```
dbgcmd

kd> !irql
Debugger saved IRQL for processor 0x0 -- 12 (SYNCH_LEVEL) [Synchronization
level]
```

!isainfo

Article • 10/25/2023

The **!isainfo** extension displays information about PNPISA cards or devices present in the system..

```
dbgcmd
```

```
!isainfo [Card]
```

Parameters

Card

Specifies a PNPISA Card. If *Card* is 0 or omitted, all the devices and cards on the PNPISA (that is, the PC I/O) Bus are displayed.

DLL

Kdexts.dll

Remarks

Here is an example of the output from this extension:

```
dbgcmd
```

```
0: kd> !isainfo
ISA PnP FDO @ 0x867b9938, DevExt @ 0x867b99f0, Bus # 0
Flags (0x80000000) DF_BUS

ISA PnP PDO @ 0x867B9818, DevExt @ 0x86595388
Flags (0x40000818) DF_ENUMERATED, DF_ACTIVATED,
DF_REQ_TRIMMED, DF_READ_DATA_PORT
```

!isr

Article • 04/03/2024

The !isr extension displays the Itanium Interruption Status Register (ISR) at the specified address.

dbgcmd

```
!isr Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Expression

Specifies the hexadecimal address of the ISR register to display. The expression @isr can also be used for this parameter. In that case, information about the current processor ISR register is displayed.

DisplayLevel

Can be any one of the following options:

0

Displays only the values of each ISR field. This is the default.

1

Displays detailed information about ISR fields that are not reserved or ignored.

2

Displays details about all ISR fields, including those that are ignored or reserved.

DLL

Kdexts.dll

This extension command can only be used with an Itanium target computer.

Remarks

Here are a couple of examples of output from this extension:

dbgcmd

```
kd> !isr @isr
isr:ed ei so ni ir rs sp na r w x vector code
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

kd> !isr @isr 2

cod : 0 : interruption Code
vec : 0 : IA32 exception vector number
rv : 0 : reserved0
x : 0 : eXecute exception
w : 0 : Write exception
r : 0 : Read exception
na : 0 : Non-Access exception
sp : 0 : Speculative load exception
rs : 0 : Register Stack
ir : 0 : Invalid Register frame
ni : 0 : Nested Interruption
so : 0 : IA32 Supervisor Override
ei : 0 : Exception IA64 Instruction
ed : 0 : Exception Deferral
rv : 0 : reserved1
```

Feedback

Was this page helpful?

 Yes

 No

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

!ivt

Article • 04/03/2024

The !ivt extension displays the Itanium interrupt vector table.

```
dbgcmd  
    !ivt [-v] [-a] [Vector]  
    !ivt -?
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Vector

Specifies an interrupt vector table entry for the current processor. If *Vector* is omitted, the entire interrupt vector table for the current processor on the target computer is displayed. Interrupt vectors that have not been assigned are not displayed unless the **-a** option is specified.

-a

Displays all interrupt vectors, including those that are unassigned.

-v

Displays detailed output.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

This extension command can only be used with an Itanium target computer.

Additional Information

For more information about how to display the interrupt dispatch tables on an x64 or x86 target computer, see [!idt](#).

Remarks

Here is an example of the output from this extension:

```
dbgcmd

kd> !ivt

Dumping IA64 IVT:

00:e000000083005f60 nt!KiPassiveRelease
0f:e000000083576830 hal!HalpPCIISALine2Pin
10:e0000000830067f0 nt!KiApcInterrupt
20:e000000083006790 nt!KiDispatchInterrupt
30:e000000083576b30 hal!HalpCMCIHandler
31:e000000083576b20 hal!HalpCPEIHandler
41:e000000085039680 i8042prt!I8042KeyboardInterruptService (KINTERRUPT
e000000085039620)
51:e000000085039910 i8042prt!I8042MouseInterruptService (KINTERRUPT
e0000000850398b0)
61:e0000000854484f0 VIDEOPRT!pVideoPortInterrupt (KINTERRUPT
e000000085448490)
71:e0000000856c9450 NDIS!ndisMIsr (KINTERRUPT e0000000856c93f0)
81:e0000000857fd000 SCSIPORT!ScsiPortInterrupt (KINTERRUPT e0000000857fcfa0)
91:e0000000857ff510 atapi!IdePortInterrupt (KINTERRUPT e0000000857ff4b0)
a1:e0000000857d84b0 atapi!IdePortInterrupt (KINTERRUPT e0000000857d8450)
a2:e0000165fff2cab0 portcls!CInterruptSyncServiceRoutine (KINTERRUPT
e0000165fff2ca50)
b1:e0000000858c7460 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT
e0000000858c7400)
b2:e0000000850382e0 USBPORT!USBPORT_InterruptService (KINTERRUPT
e000000085038280)
d0:e0000000835768d0 hal!HalpClockInterrupt
e0:e000000083576850 hal!HalpIpiInterruptHandler
f0:e0000000835769c0 hal!HalpProfileInterrupt
f1:e000000083576830 hal!HalpPCIISALine2Pin
fd:e000000083576b10 hal!HalpMcRzHandler
fe:e000000083576830 hal!HalpPCIISALine2Pin
```

Feedback

Was this page helpful?

 Yes

 No

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

!job

Article • 10/25/2023

The **!job** extension displays a job object.

dbgcmd

```
!job [Process [Flags]]
```

Parameters

Process

Specifies the hexadecimal address of a process or a thread whose associated job object is to be examined. If this is omitted, or equal to zero, the job associated with the current process is displayed.

Flags

Specifies what the display should contain. This can be a sum of any of the following bit values. The default is 0x1:

Bit 0 (0x1)

Causes the display to include job settings and statistics.

Bit 1 (0x2)

Causes the display to include a list of all processes in the job.

DLL

Kdexts.dll

Additional Information

For information about job objects, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension:

dbgcmd

```
kd> !process 52c
Searching for Process with Cid == 52c
PROCESS 8276c550 SessionId: 0 Cid: 052c Peb: 7ffdf000 ParentCid: 0060
    DirBase: 01289000 ObjectTable: 825f0368 TableSize: 24.
    Image: cmd.exe
    VadRoot 825609e8 Vads 30 Clone 0 Private 77. Modified 0. Locked 0.
    DeviceMap e1733f38
        Token e1681610
        ElapsedTime 0:00:12.0949
        UserTime 0:00:00.0359
        ....
        CommitCharge 109
        Job 8256e1f0
```

```
kd> !job 8256e1f0
Job at ffffffff8256e1f0
    TotalPageFaultCount 0
    TotalProcesses 1
    ActiveProcesses 1
    TotalTerminatedProcesses 0
    LimitFlags 0
    MinimumWorkingSetSize 0
    MaximumWorkingSetSize 0
    ActiveProcessLimit 0
    PriorityClass 0
    UIRestrictionsClass 0
    SecurityLimitFlags 0
    Token 00000000
```

!kb, !kv

Article • 03/25/2024

The !kb and !kv extension commands are obsolete. Use the [kb \(Display Stack Backtrace\)](#) and [kv \(Display Stack Backtrace\)](#) commands instead.

Feedback

Was this page helpful?

 Yes

 No

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

!loadermemorylist

Article • 04/03/2024

The **!loadermemorylist** extension displays the memory allocation list that the Windows boot loader passes to Windows.

```
dbgcmd
```

```
!loadermemorylist ListHeadAddress
```

Parameters

ListHeadAddress

Specifies the address of a list header.

DLL

Kdexts.dll

Remarks

This extension is designed to be used at the beginning of the system boot process while Ntldr is running. It displays a memory allocation list that includes the start, end, and type of each page range.

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!lockedpages

Article • 10/25/2023

The **!lockedpages** extension displays driver-locked pages for a specified process.

Syntax

```
dbgcmd
```

```
!lockedpages [Process]
```

Parameters

Process

Specifies a process. If *Process* is omitted, the current process is used.

DLL

Kdexts.dll

Remarks

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!locks (!kdext*.locks)

Article • 10/25/2023

The **!locks** extension in Kdextx86.dll and Kdexts.dll displays information about kernel ERESOURCE locks.

This extension command should not be confused with the [!ntsdexsts.locks](#) extension command.

```
dbgcmd
```

```
!locks [Options] [Address]
```

Parameters

Options Specifies the amount of information to be displayed. Any combination of the following options can be used:

-v

Displays detailed information about each lock.

-p

Display all available information about the locks, including performance statistics.

-d

Display information about all locks. Otherwise, only locks with contention are displayed.)

Address

Specifies the hexadecimal address of the ERESOURCE lock to be displayed. If *Address* is 0 or omitted, information about all ERESOURCE locks in the system will be displayed.

DLL

Kdexts.dll

Remarks

The **!locks** extension displays all locks held on resources by threads. A lock can be shared or exclusive, which means no other threads can gain access to that resource. This information is useful when a deadlock occurs on a system. A deadlock is caused by one

non-executing thread holding an exclusive lock on a resource that the executing thread needs.

You can usually pinpoint a deadlock in Microsoft Windows 2000 by finding one non-executing thread that holds an exclusive lock on a resource that is required by an executing thread. Most of the locks are shared.

Here is an example of the basic **!locks** output:

```
dbgcmd

kd> !locks
***** DUMP OF ALL RESOURCE OBJECTS *****
KD: Scanning for held locks.....
Resource @ 0x80e97620      Shared 4 owning threads
    Threads: ff688da0-01<*> ff687da0-01<*> ff686da0-01<*> ff685da0-01<*>
KD: Scanning for held
locks..... .

Resource @ 0x80e23f38      Shared 1 owning threads
    Threads: 80ed0023-01<*> *** Actual Thread 80ed0020
KD: Scanning for held locks.

Resource @ 0x80d8b0b0      Shared 1 owning threads
    Threads: 80ed0023-01<*> *** Actual Thread 80ed0020
2263 total locks, 3 locks currently held
```

Note that the address for each thread displayed is followed by its thread count (for example, "-01"). If a thread is followed by "<*>", that thread is one of the owners of the lock. In some instances, the initial thread address contains an offset. In that case, the actual thread address is displayed as well.

If you want to find more information about one of these resource objects, use the address that follows "Resource @" as an argument for future commands. To investigate the second resource shown in the preceding example, you could use **dt ERESOURCE 80d8b0b0** or **!thread 80ed0020**. Or you could use the **!locks** extension again with the **-v** option:

```
dbgcmd

kd> !locks -v 80d8b0b0
Resource @ 0x80d8b0b0      Shared 1 owning threads
    Threads: 80ed0023-01<*> *** Actual Thread 80ed0020
    THREAD 80ed0020  Cid 4.2c  Teb: 00000000 Win32Thread: 00000000 WAIT:
(WrQueue) KernelMode Non-Alertable
```

```
8055e100 Unknown
Not impersonating
GetUlongFromAddress: unable to read from 00000000
    Owning Process 80ed5238
    WaitTime (ticks)        44294977
    Context Switch Count   147830
    UserTime                0:00:00.0000
    KernelTime              0:00:02.0143
    Start Address nt!ExpWorkerThread (0x80506aa2)
    Stack Init fafa4000 Current fafa3d18 Base fafa4000 Limit fafa1000 Call
0
    Priority 13 BasePriority 13 PriorityDecrement 0
ChildEBP RetAddr
fafa3d30 804fe997 nt!KiSwapContext+0x25 (FPO: [EBP 0xfafa3d48] [0,0,4])
[D:\NT\base\ntos\ke\i386\ctxswap.asm @ 139]
fafa3d48 80506a17 nt!KiSwapThread+0x85 (FPO: [Non-Fpo]) (CONV: fastcall)
[d:\nt\base\ntos\ke\thredsup.c @ 1960]
fafa3d78 80506b36 nt!KeRemoveQueue+0x24c (FPO: [Non-Fpo]) (CONV: stdcall)
[d:\nt\base\ntos\ke\queueobj.c @ 542]
fafa3dac 805ad8bb nt!ExpWorkerThread+0xc6 (FPO: [Non-Fpo]) (CONV: stdcall)
[d:\nt\base\ntos\ex\worker.c @ 1130]
fafa3ddc 8050ec72 nt!PspSystemThreadStartup+0x2e (FPO: [Non-Fpo]) (CONV:
stdcall) [d:\nt\base\ntos\ps\create.c @ 2164]
00000000 00000000 nt!KiThreadStartup+0x16
[D:\NT\base\ntos\ke\i386\threadbg.asm @ 81]

1 total locks, 1 locks currently held
```

!logonsession

Article • 04/03/2024

The **!logonsession** extension displays information about a specified logon session.

Free Build Syntax

```
dbgcmd  
!logonsession LUID
```

Checked Build Syntax

```
dbgcmd  
!logonsession LUID [InfoLevel]
```

Parameters

LUID

Specifies the locally unique identifier (LUID) of a logon session to display. If *LUID* is 0, information about all logon sessions is displayed.

To display information about the system session and all system tokens in a checked build, enter **!logonsession 3e7 1**. Checked builds were available on older versions of Windows before Windows 10, version 1803.

InfoLevel

(Checked Build Only) Specifies how much token information is displayed. The *InfoLevel* parameter can take values from 0 to 4, with 0 representing the least information and 4 representing the most information. Checked builds were available on older versions of Windows before Windows 10, version 1803.

DLL

Kdexts.dll

Additional Information

For information about logon sessions, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example of the output from this extension on a free build:

```
dbgcmd

kd> !logonsession 0

Dumping all logon sessions.

** Session 0 = 0x0
    LogonId      = {0x0 0x0}
    References   = 0
** Session 1 = 0x8ebb50
    LogonId      = {0xe9f1 0x0}
    References   = 21
** Session 2 = 0x6e31e0
    LogonId      = {0x94d1 0x0}
    References   = 1
** Session 3 = 0x8ecd60
    LogonId      = {0x6b31 0x0}
    References   = 0
** Session 4 = 0xe0000106
    LogonId      = {0x0 0x0}
    References   = 0
** Session 5 = 0x0
    LogonId      = {0x0 0x0}
    References   = 0
** Session 6 = 0x8e9720
    LogonId      = {0x3e4 0x0}
    References   = 6
** Session 7 = 0xe0000106
    LogonId      = {0x0 0x0}
    References   = 0
** Session 8 = 0xa2e160
    LogonId      = {0x3e5 0x0}
    References   = 3
** Session 9 = 0xe0000106
    LogonId      = {0x0 0x0}
    References   = 0
** Session 10 = 0x3ca0
    LogonId      = {0x3e6 0x0}
    References   = 2
** Session 11 = 0xe0000106
    LogonId      = {0x0 0x0}
    References   = 0
** Session 12 = 0x1cd0
    LogonId      = {0x3e7 0x0}
    References   = 33
```

```
** Session 13 = 0xe0000106
LogonId      = {0x0 0x0}
References   = 0
14 sessions in the system.
```

You can stop execution at any point by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!lookaside

Article • 10/25/2023

The **!lookaside** extension displays information about look-aside lists, resets the counters of look-aside lists, or modifies the depth of a look-aside list.

dbgcmd

```
!lookaside [Address [Options [Depth]]]
!lookaside [-all]
!lookaside 0 [-all]
```

Parameters

Address

Specifies the hexadecimal address of a look-aside list to be displayed or modified.

If *Address* is omitted (or 0) and the **-all** option is not specified, a set of well-known, standard system look-aside lists is displayed. The set of lists is not exhaustive; that is, it does not include all system look-aside lists. Also, the set does not include custom look-aside lists that were created by calls to [ExInitializePagedLookasideList](#) or [ExInitializeNPagedLookasideList](#).

If *Address* is omitted (or 0) and the **-all** option is specified, all look-aside lists are displayed.

Options Controls what operation will be taken. The following possible *Options* are supported. The default is zero:

0

Displays information about the specified look-aside list or lists.

1

Resets the counters of the specified look-aside list.

2

Modifies the depth of the specified look-aside list. This option can only be used if *Address* is nonzero.

Depth

Specifies the new maximum depth of the specified look-aside list. This parameter is permitted only if *Address* is nonzero and *Options* is equal to 2.

Additional Information

For information about look-aside lists, see the [Using Lookaside Lists](#) and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Look-aside lists are multiprocessor-safe mechanisms for managing pools of fixed-size entries from either paged or nonpaged memory.

Look-aside lists are efficient, because the routines do not use spin locks on most platforms.

Note that if the current depth of a look-aside list exceeds the maximum depth of that list, then freeing a structure associated with that list will result in freeing it into pool memory, rather than list memory.

Here is an example of the output from this extension:

```
dbgcmd

!lookaside 0xfffff88001294f80

Lookaside "" @ 0xfffff88001294f80 Tag(hex): 0x7366744e "Ntfs"
  Type          =      0011 PagedPool RaiseIfAllocationFailure
  Current Depth =          0 Max Depth   =          4
  Size          =       496 Max Alloc    =      1984
  AllocateMisses =         8 FreeMisses  =          0
  TotalAllocates =     272492 TotalFrees  =     272488
  Hit Rate      =      99% Hit Rate   =      100%
```

Requirements

DLL	Kdexts.dll
-----	------------

!lpc

Article • 10/25/2023

Important Lpc is now emulated in alpc, use the !alpc extension instead.

The **!lpc** extension displays information about all local procedure call (LPC) ports and messages in the target system.

dbgcmd

```
!lpc message MessageID
!lpc port Port
!lpc scan Port
!lpc thread Thread
!lpc PoolSearch
!lpc
```

Parameters

message

(Windows Server 2003, Windows XP, and Windows 2000 only) Displays information about a message, such as the server port that contains the message in the queue, and the thread waiting for this message, if any.

MessageID

(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the message ID of the message to be displayed. If the value of this parameter is 0 or this parameter is omitted, the **!lpc message** command displays a summary list of messages. (In Windows 2000 with Service Pack 1 (SP1), the summary includes all messages in the LPC zone. In Windows 2000 with Service Pack 2 (SP2), Windows XP, and later versions of Windows, the summary includes all messages in the kernel pool. Paged-out messages are not included.)

port

(Windows Server 2003, Windows XP, and Windows 2000 only) Displays port information, such as the name of the port, its semaphore state, messages in its queues, threads in its rundown queue, its handle count, its references, and related ports.

scan

(Windows Server 2003, Windows XP, and Windows 2000 only) Displays summary information about the specified port and about all ports that are connected to it.

Port

(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the hexadecimal address of the port to be displayed. If the **!lpc port** command is used, and *Port* is 0 or is omitted, a summary list of all LPC ports is displayed. If the **!lpc scan** command is used, *Port* must specify the address of an actual port.

thread

(Windows Server 2003, Windows XP, and Windows 2000 only) Displays port information for all ports that contain the specified thread in their rundown port queues.

Thread

(Windows Server 2003, Windows XP, and Windows 2000 only) Specifies the hexadecimal address of the thread. If this is 0 or omitted, the **!lpc thread** command displays a summary list of all threads that are performing any LPC operations.

PoolSearch

(Windows Server 2003 and Windows XP only) Determines whether the **!lpc message** command searches for messages in the kernel pool. Each time **!lpc PoolSearch** is used, this setting toggles on or off (the initial setting is to not search the kernel pool). This only affects **!lpc message** commands that specify a nonzero value for *MessageID*.

DLL

Windows 2000	Kdextx86.dll
Windows XP	Kdexts.dll
Windows Server 2003	

Additional Information

For information about LPCs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is not supported in Windows Vista and later versions of Windows.

In Windows Server 2003, Windows XP, and Windows 2000, using **!lpc** with no arguments displays help for this extension in the Debugger Command window.

If you have a thread that is marked as waiting for a reply to a message, use the **!lpc message** command with the ID of the delayed message. This command displays the specified message, the port that contains it, and all related threads.

If the message is not found and there were no read errors (such as "Unable to access zone segment"), the server received the message.

In this case, the server port can usually be found by using the **!lpc thread** command. Threads that are waiting for replies are linked into a server communication queue. This command will display all ports that contain the specified thread. After you know the port address, use the **!lpc port** command. More specific information about each thread can then be obtained by using the **!lpc thread** command with the address of each thread.

Here are several examples of the output from this extension from a Windows XP system:

In this example, all port LPC ports are displayed.

```
dbgcmd

kd> !lpc port
Scanning 225 objects
    1 Port: 0xe1405650 Connection: 0xe1405650 Communication: 0x00000000
'SeRmCommandPort'
    1 Port: 0xe141ef50 Connection: 0xe141ef50 Communication: 0x00000000
'SmApiPort'
    1 Port: 0xe13c5740 Connection: 0xe13c5740 Communication: 0x00000000
'ApiPort'
    1 Port: 0xe13d9550 Connection: 0xe13d9550 Communication: 0x00000000
'SbApiPort'
    3 Port: 0xe13d8830 Connection: 0xe141ef50 Communication: 0xe13d8910
'
80000004 Port: 0xe13d8910 Connection: 0xe141ef50 Communication: 0xe13d8830
'
    3 Port: 0xe13d8750 Connection: 0xe13d9550 Communication: 0xe13a4030
'
....
```

In the previous example, the port at address e14ae238 has no messages; that is, all messages have been picked up and no new messages have arrived.

```
dbgcmd

kd> !lpc port e14ae238
Server connection port e14ae238 Name: ApiPort
Handles: 1 References: 107
    Server process : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
    Semaphore state 0 (0x0)
```

```
The message queue is empty
The LpcDataInfoChainHead queue is empty
```

In the previous example, the port at 0xe14ae238 has messages which have been queued, but not yet picked up by the server.

```
dbgcmd

kd> !lpc port 0xe14ae238

Server connection port e14ae238 Name: ApiPort
Handles: 1 References: 108
    Server process : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
Semaphore state 0 (0x0)
    Messages in queue:
0000 e20d9b80 - Busy Id=0002249c From: 0584.0680 Context=00000021
[e14ae248 . e14ae248]
Length=0098007c Type=00000001 (LPC_REQUEST)
    Data: 00000000 0002021e 00000584 00000680 002f0001
00000007
    The message queue contains 1 messages
    The LpcDataInfoChainHead queue is empty
```

The remaining Windows XP examples concern the other options that can be used with this extension.

```
dbgcmd

kd> !lpc message 222be
Searching message 222be in threads ...
Client thread 842a4db0 waiting a reply from 222be
Searching thread 842a4db0 in port rundown queues ...

Server communication port 0xe114a3c0
Handles: 1 References: 1
The LpcDataInfoChainHead queue is empty
Connected port: 0xe1e7b948      Server connection port: 0xe14ae238

Client communication port 0xe1e7b948
Handles: 1 References: 3
The LpcDataInfoChainHead queue is empty

Server connection port e14ae238 Name: ApiPort
Handles: 1 References: 107
    Server process : 84aa0140 (csrss.exe)
    Queue semaphore : 84a96da8
Semaphore state 0 (0x0)
    The message queue is empty
```

```
The LpcDataInfoChainHead queue is empty  
Done.
```

```
dbgcmd
```

```
kd> !lpc thread 842a4db0  
Searching thread 842a4db0 in port rundown queues ...  
  
Server communication port 0xe114a3c0  
Handles: 1 References: 1  
The LpcDataInfoChainHead queue is empty  
Connected port: 0xe1e7b948 Server connection port: 0xe14ae238  
  
Client communication port 0xe1e7b948  
Handles: 1 References: 3  
The LpcDataInfoChainHead queue is empty  
  
Server connection port e14ae238 Name: ApiPort  
Handles: 1 References: 107  
Server process : 84aa0140 (csrss.exe)  
Queue semaphore : 84a96da8  
Semaphore state 0 (0x0)  
The message queue is empty  
The LpcDataInfoChainHead queue is empty
```

```
dbgcmd
```

```
kd> !lpc scan e13d8830  
Scanning 225 objects  
3 Port: 0xe13d8830 Connection: 0xe141ef50 Communication: 0xe13d8910  
'  
80000004 Port: 0xe13d8910 Connection: 0xe141ef50 Communication: 0xe13d8830  
'  
Scanning 3 objects
```

!mca

Article • 10/25/2023

The !mca extension displays the machine check architecture (MCA) registers.

```
dbgcmd
```

```
!mca
```

Parameters

Address

(Itanium target only) Specifies the address of the MCA error record.

Flags

(Itanium target only) Specifies the level of output. *Flags* can be any combination of the following bits. The default value is 0xFF, which displays all sections present in the log.

Bit 0 (0x1)

Displays the processor section.

Bit 1 (0x2)

Displays the platform-specific section.

Bit 2 (0x4)

Displays the memory section.

Bit 3 (0x8)

Displays the PCI component section.

Bit 4 (0x10)

Displays the PCI bus section.

Bit 5 (0x20)

Displays the SystemEvent Log section.

Bit 6 (0x40)

Displays the platform host controller section.

Bit 7 (0x80)

Displays to include the platform bus section.

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

Remarks

On an x86 target, **!mca** displays the machine check registers supported by the active processor. It also displays basic CPU information (identical to that displayed by [!cpuinfo](#)). Here is an example of the output from this extension:

```
dbgcmd

0: kd> !mca
MCE: Enabled, Cycle Address: 0x00000001699f7a00, Type: 0x0000000000000000

MCA: Enabled, Banks 5, Control Reg: Supported, Machine Check: None.
Bank Error Control Register Status Register
 0. None 0x000000000000007f 0x0000000000000000
 1. None 0x00000000ffffffff 0x0000000000000000
 2. None 0x000000000000ffff 0x0000000000000000
 3. None 0x0000000000000007 0x0000000000000000
 4. None 0x0000000000003fff 0x0000000000000000

No register state available.

CP F/M/S Manufacturer MHz Update Signature Features
0 15,5,0 SomeBrandName 1394 0000000000000000 a0017fff
```

Note that this extension requires private HAL symbols. Without these symbols, the extension will display the message "HalpFeatureBits not found" along with basic CPU information. For example:

```
dbgcmd

kd> !mca
HalpFeatureBits not found
CP F/M/S Manufacturer MHz Update Signature Features
0 6,5,1 GenuineIntel 334 0000004000000000 00001fff
```

!memlist

Article • 04/03/2024

The **!memlist** extension scans physical memory lists from the page frame number (PFN) database in order to check them for consistency.

Syntax

```
!memlist Flags
```

Parameters

Flags

Specifies which memory lists to verify. At present, only one value has been implemented:

Bit 0 (0x1)

Causes the zeroed pages list to be verified.

DLL

Kdexts.dll

Remarks

This extension will only check the zeroed pages list to make sure that all pages in that list are zeroed. The appropriate syntax is:

```
dbgcmd
```

```
kd> !memlist 1
```

Feedback

Was this page helpful?

 Yes

 No

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

!memusage

Article • 12/12/2023

The **!memusage** extension displays summary statistics about physical memory use.

Syntax

```
!memusage [Flags]
```

Parameters

Flags

Can be any one of the following values. The default is 0x0.

0x0

Displays general summary information, along with a more detailed description of the pages in the PFN database. See the Remarks section for an example of this type of output.

0x1

Displays only summary information about the modified no-write pages in the PFN database.

0x2

Displays only detailed information about the modified no-write pages in the PFN database.

0x8

Displays only general summary information about memory use.

Environment

Modes: kernel mode only

DLL

Kdexts.dll

Additional Information

Physical memory statistics are collected from the Memory Manager's page frame number (PFN) database table.

This command takes a long time to run, especially if the target computer is running in 64-bit mode, due to the greater amount of data to obtain. While it is loading the PFN database, a counter shows its progress. To speed up this loading, use a network connection, or increase the COM port speed with the [CTRL+A \(Toggle Baud Rate\)](#) key, or use the [.cache \(Set Cache Size\)](#) command to increase the cache size (perhaps to around 10 MB).

The `!memusage` command can also be used while performing [local kernel debugging](#).

Here is an example of the output from this extension:

```
dbgcmd

kd> !memusage
loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
    Zeroed:      218 (     872 kb)
    Free:        831 (   3324 kb)
    Standby:  124049 ( 496196 kb)
    Modified:   55101 ( 220404 kb)
    ModifiedNoWrite: 58 (    232 kb)
    Active/Valid: 321846 ( 1287384 kb)
    Transition:    8 (     32 kb)
    SLIST/Temp:   1533 (    6132 kb)
    Bad:          0 (      0 kb)
    Unknown:      0 (      0 kb)
    TOTAL:      503644 ( 2014576 kb)

Dangling Yes Commit:      184 (     736 kb)
Dangling No Commit:  81706 ( 326824 kb)
Building kernel map
Finished building kernel map
Scanning PFN database - (100% complete)

...
```

Also included in the report is detailed information about the usage of memory that is visible to the debugger.

```
dbgcmd

Usage Summary (in Kb):
Control      Valid  Standby  Dirty  Shared  Locked  PageTables  name
fffffaf0fb369f010  204     956      0     32    204      0 mapped_file(
```

```
shell32.dll )  
fffffaf0fb369f270 492 60 0 252 492 0 mapped_file(  
KernelBase.dll )  
fffffaf0fb36ad050 20 36 0 0 20 0 mapped_file(  
WMIsvc.dll )  
fffffaf0fb36adad0 88 144 0 40 88 0 mapped_file( Can't  
read file name buffer at fffffc10e0497e170 )  
fffffaf0fb36b5670 780 1012 0 560 780 0 mapped_file(  
KernelBase.dll )  
fffffaf0fb36b5910 44 144 0 28 44 0 mapped_file(  
cfgmgr32.dll )  
fffffaf0fb36bc270 8 0 0 0 8 0 mapped_file( Can't  
read file name buffer at fffffc10e061a17d0 )  
fffffaf0fb36bc520 24 56 0 4 24 0 mapped_file(  
ShareHost.dll )
```

...

The first column displays the address of the control area structure that describes each mapped structure. Use the [!ca](#) extension command to display these control areas.

Remarks

You can use the [!lvm](#) extension command to analyze virtual memory use. This extension is typically more useful than [!memusage](#). For more information about memory management, see *Microsoft Windows Internals*, by Pavel Yosifovich, Andrea Allievi, Alex Ionescu, Mark Russinovich and David Solomon.

The [!pfn](#) extension command can be used to display a particular page frame entry in the PFN database.

The [!pool](#) extension displays information about a specific pool allocation or about the entire system-wide pool.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!mps

Article • 10/25/2023

The **!mps** extension displays BIOS information for the Intel Multiprocessor Specification (MPS) of the target computer.

dbgcmd

!mps [Address]

Parameters

Address

Specifies the hexadecimal address of the MPS table in the BIOS. If this is omitted, the information is obtained from the HAL. This will require HAL symbols.

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

Additional Information

For more information about BIOS debugging, see [Debugging BIOS Code](#). For more information about the MPS, refer to the appropriate version of the Intel MultiProcessor Specification.

!mtrr

Article • 10/25/2023

The **!mtrr** extension displays the contents of the MTRR register.

```
dbgcmd
```

```
!mtrr
```

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

!npx

Article • 04/03/2024

The **!npx** extension displays the contents of the floating-point register save area.



Parameters

Address

Specifies the hexadecimal address of the FLOATING_SAVE_AREA structure.

DLL

Kdexts.dll

Remarks

This extension command can only be used with an x86-based target computer.

Feedback

Was this page helpful?

Yes

No

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

!ob, !od, !ow

Article • 03/27/2024

The **!ob**, **!od**, and **!ow** extension commands are obsolete. Use the [ob, od, ow \(Output to Port\)](#) commands instead.

Feedback

Was this page helpful?

 Yes

 No

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

!object

Article • 10/25/2023

The **!object** extension displays information about a system object.

```
dbgcmd  
!object Address [Flags]  
!object Path  
!object 0 Name  
!object -p  
!object {-h|-?}
```

Parameters

Address

If the first argument is a nonzero hexadecimal number, it specifies the hexadecimal address of the system object to be displayed.

Flags

Specifies the level of detail in the command output.

Set *Flags* to a bitwise OR of these values:

0x0

Display object type.

0x1

Display object type, object name, and reference counts.

0x8

Display the contents of an object directory or the target of a symbolic link. This flag has an effect only if **0x1** is also set.

0x10

Display optional object headers.

0x20

Display the full path to a named object. This flag has an effect only if **0x1** is also set.

The *Flags* parameter is optional. The default value is **0x9**.

Path

If the first argument begins with a backslash (), **!object** interprets it as an object path

name. When this option is used, the display will be arranged according to the directory structure used by the Object Manager.

Name

If the first argument is zero, the second argument is interpreted as the name of a class of system objects for which to display all instances.

-p

Display private object name spaces.

{-h|-?}

Display help for this command.

DLL

Kdexts.dll

Examples

This example passes the path of the \Device directory to !object. The output lists all objects in the \Device directory.

```
dbgcmd

0: kd> !object \Device
Object: fffffc00b074166a0  Type: (fffffe0083b768690) Directory
    ObjectHeader: fffffc00b07416670 (new version)
    HandleCount: 0  PointerCount: 224
    Directory Object: fffffc00b074092e0  Name: Device

    Hash Address          Type      Name
    ----  -----          ----      ----
    00   fffffe0083e6a61f0 Device    00000044
        fffffe0083dcc4050 Device    00000030
        fffffe0083d34f050 Device    NDMP2
        fffffe0083bdf7060 Device    NTPNP_PCI0002
        ...
        fffffe0083b85d060 Device    USBPDO-8
        fffffe0083d33d050 Device    USBFDO-6
        ...
        fffffe0083bdf0060 Device    NTPNP_PCI0001
```

Choose one of listed objects, say USBPDO-8. Pass the address of USBPDO-8 (fffffe0083b85d060) to !object. Set *Flags* to 0x0 to get minimal information.

```
dbgcmd
```

```
0: kd> !object fffffe0083b85d060 0x0
Object: fffffe0083b85d060  Type: (fffffe0083b87df20) Device
    ObjectHeader: fffffe0083b85d030 (new version)
```

Include name and reference count information for the same object by setting *Flags* to 0x1.

```
dbgcmd

0: kd> !object fffffe0083b85d060 0x1
Object: fffffe0083b85d060  Type: (fffffe0083b87df20) Device
    ObjectHeader: fffffe0083b85d030 (new version)
    HandleCount: 0  PointerCount: 6
    Directory Object: fffffc00b074166a0  Name: USBPDO-8
```

Get optional header information for the same object by setting *Flags* to 0x10.

```
dbgcmd

0: kd> !object fffffe0083b85d060 0x10
Object: fffffe0083b85d060  Type: (fffffe0083b87df20) Device
    ObjectHeader: fffffe0083b85d030 (new version)
Optional Headers:
    NameInfo(fffffe0083b85d010)
```

The following example calls **!object** twice for a Directory object. The first time, the contents of the directory are not displayed because the 0x8 flag is not set. The second time, the contents of the directory are displayed because both the 0x8 and 0x1 flags are set (Flags = 0x9).

```
dbgcmd

0: kd> !object fffffc00b07481d00 0x1
Object: fffffc00b07481d00  Type: (fffffe0083b768690) Directory
    ObjectHeader: fffffc00b07481cd0 (new version)
    HandleCount: 0  PointerCount: 3
    Directory Object: fffffc00b07481eb0  Name: Filters

0: kd> !object fffffc00b07481d00 0x9
Object: fffffc00b07481d00  Type: (fffffe0083b768690) Directory
    ObjectHeader: fffffc00b07481cd0 (new version)
    HandleCount: 0  PointerCount: 3
    Directory Object: fffffc00b07481eb0  Name: Filters
```

Hash	Address	Type	Name
-----	-----	-----	-----

19	fffffe0083c5f56e0	Device	FltMgrMsg
21	fffffe0083c5f5060	Device	FltMgr

The following example calls **!object** twice for a SymbolicLink object. The first time, the target of the symbolic link is not displayed because the 0x8 flag is not set. The second time, the target of the symbolic link is splayed because both the 0x8 and 0x1 flags are set (Flags = 0x9).

```
dbgcmd
```

```
0: kd> !object fffffc00b07628fb0 0x1
Object: fffffc00b07628fb0  Type: (fffffe0083b769450) SymbolicLink
    ObjectHeader: fffffc00b07628f80 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: fffffc00b074166a0  Name: Ip6

0: kd> !object fffffc00b07628fb0 0x9
Object: fffffc00b07628fb0  Type: (fffffe0083b769450) SymbolicLink
    ObjectHeader: fffffc00b07628f80 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: fffffc00b074166a0  Name: Ip6
    Target String is '\Device\Tdx'
```

Additional Information

For information about objects and the object manager, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

See also

[Object Reference Tracing](#)

[!obtrace](#)

[!handle](#)

[Determining the ACL of an Object](#)

[Kernel-Mode Extension Commands](#)

!obtrace

Article • 04/03/2024

The **!obtrace** extension displays object reference tracing data for the specified object.

```
dbgcmd
```

```
!obtrace Object
```

Parameters

Object

A pointer to the object or a path.

DLL

Kdexts.dll

Additional Information

For more information about the Global Flags utility (GFlags), see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The object reference tracing feature of Windows records sequential stack traces whenever an object reference counter is incremented or decremented.

Before using this extension to display object reference tracing data, you must use [GFlags](#) to enable [object reference tracing](#) for the specified object. You can enable object reference tracing as a kernel flag (run-time) setting, in which the change is effective immediately, but disappears if you shut down or restart; or as a registry setting, which requires a restart, but remains effective until you change it.

Here is an example of the output from the **!obtrace** extension:

```
dbgcmd
```

```

kd> !obtrace 0xfa96f700
Object: fa96f700           Image: cmd.exe
Sequence (+/-) Stack
-----
2421d    +1  nt!ObCreateObject+180
          nt!NtCreateEvent+92
          nt!KiFastCallEntry+104
          nt!ZwCreateEvent+11
          win32k!UserThreadCallout+6f
          win32k!W32pThreadCallout+38
          nt!PsConvertToGuiThread+174
          nt!KiBBTUnexpectedRange+c

2421e    -1  nt!ObfDereferenceObject+19
          nt!NtCreateEvent+d4
          nt!KiFastCallEntry+104
          nt!ZwCreateEvent+11
          win32k!UserThreadCallout+6f
          win32k!W32pThreadCallout+38
          nt!PsConvertToGuiThread+174
          nt!KiBBTUnexpectedRange+c

2421f    +1  nt!ObReferenceObjectByHandle+1c3
          win32k!xxxCreateThreadInfo+37d
          win32k!UserThreadCallout+6f
          win32k!W32pThreadCallout+38
          nt!PsConvertToGuiThread+174
          nt!KiBBTUnexpectedRange+c

24220    +1  nt!ObReferenceObjectByHandle+1c3
          win32k!ProtectHandle+22
          win32k!xxxCreateThreadInfo+3a0
          win32k!UserThreadCallout+6f
          win32k!W32pThreadCallout+38
          nt!PsConvertToGuiThread+174
          nt!KiBBTUnexpectedRange+c

24221    -1  nt!ObfDereferenceObject+19
          win32k!xxxCreateThreadInfo+3a0
          win32k!UserThreadCallout+6f
          win32k!W32pThreadCallout+38
          nt!PsConvertToGuiThread+174
          nt!KiBBTUnexpectedRange+c

```

References: 3, Dereferences 2

The primary indicators in the `!obtrace 0xfa96f700` display are listed in the following table.

[] Expand table

Parameter	Meaning
Sequence	Represents the order of operations.
+/-	<p>Indicates a reference or a dereference operation.</p> <p>+1 indicates a reference operation.</p> <p>-1 indicates a dereference operation.</p> <p>+/- n indicates multiple reference/dereference operations.</p>

The object reference traces on x64-based target computers might be incomplete because it is not always possible to acquire stack traces at IRQL levels higher than PASSIVE_LEVEL.

You can stop execution at any time by pressing CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!openmaps

Article • 10/25/2023

The **!openmaps** extension displays the referenced buffer control blocks (BCBs) and virtual address control blocks (VACBs) for the specified shared cache map.

dbgcmd

```
!openmaps Address [Flag]
```

Parameters

Address

Specifies the address of the shared cache map.

Flag

Determines which control blocks are displayed. If *Flag* is 1, the debugger displays all control blocks. If *Flag* is 0, the debugger displays only referenced control blocks. The default is 0.

DLL

Kdexts.dll

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the [!icchelp](#) extension.

!pars

Article • 04/03/2024

The **!pars** extension displays a specified processor application registers file.

```
dbgcmd
```

```
!pars Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address

Specifies the address of a processor application registers file.

DLL

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Feedback

Was this page helpful?

 Yes

 No

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

!pat

Article • 10/25/2023

The **!pat** extension displays the Page Attribute Table (PAT) registers for the target processor.

```
dbgcmd
```

```
!pat Flag  
!pat
```

Parameters

Flag

If *Flag* is set, the debugger verifies that the PAT feature is present before the PAT is displayed.

DLL

Kdexts.dll

This extension command can only be used with an x86-based target computer.

!pci

Article • 10/25/2023

The `!pci` extension displays the current status of the peripheral component interconnect (PCI) buses, as well as any devices attached to those buses.

dbgcmd

```
!pci [Flags [Segment] [Bus [Device [Function [MinAddress MaxAddress]]]]]
```

Parameters

Flags

Specifies the level of output. Can be any combination of the following bits:

Bit 0 (0x1)

Causes a verbose display.

Bit 1 (0x2)

Causes the display to include all buses in the range from bus 0 (zero) to the specified *Bus*.

Bit 2 (0x4)

Causes the display to include information in raw byte format. If *MinAddress*, *MaxAddress*, or flag bit 0x8 is set, this bit is automatically set as well.

Bit 3 (0x8)

Causes the display to include information in raw DWORD format.

Bit 4 (0x10)

Causes the display to include invalid device numbers. If *Device* is specified, this flag is ignored.

Bit 5 (0x20)

Causes the display to include invalid function numbers.

Bit 6 (0x40)

Causes the display to include capabilities.

Bit 7 (0x80)

Causes the display to include Intel 8086 device-specific information.

Bit 8 (0x100)

Causes the display to include the PCI configuration space.

Bit 9 (0x200)

Causes the display to include segment information. When this bit is included, the *Segment* parameter must be included.

Bit 10 (0x400)

Causes the display to include all valid segments in the range from segment 0 to the specified segment. When this bit is included, the *Segment* parameter must be included.

Segment

Specifies the number of the segment to be displayed. Segment numbers range from 0 to 0xFFFF. If *Segment* is omitted, information about the primary segment (segment 0) is displayed. If *Flags* includes bit 10 (0x400), *Segment* specifies the highest valid segment to be displayed.

Bus

Specifies the bus to be displayed. *Bus* can range from 0 to 0xFF. If it is omitted, information about the primary bus (bus 0) is displayed. If *Flags* includes bit 1 (0x2), *Bus* specifies the highest bus number to be displayed.

Device

Specifies the slot device number for the device. If this is omitted, information about all devices is printed.

Function

Specifies the slot function number for the device. If this is omitted, all information about all device functions is printed.

MinAddress

Specifies the first address from which raw bytes or DWORDs are to be displayed. This must be between 0 and 0xFF.

MaxAddress

Specifies the last address from which raw bytes or DWORDs are to be displayed. This must be between 0 and 0xFF, and not less than *MinAddress*.

DLL

Windows 2000	Kext.dll Kdextx86.dll
Windows XP and later	Kext.dll

This extension command can only be used with an x86-based target computer.

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command and additional examples. For information about PCI buses, see the Windows Driver Kit (WDK) documentation.

Remarks

To edit the PCI configuration space, use [!ecb](#), [!ecd](#), or [!ecw](#).

The following example displays a list of all buses and their devices. This command will take a long time to execute. You will see a moving counter at the bottom of the display while the debugger scans the target system for PCI buses:

```
dbgcmd

kd> !pci 2 ff
PCI Bus 0
00:0 8086:1237.02 Cmd[0106:.mb..s] Sts[2280:.....] Device Host bridge
0d:0 8086:7000.01 Cmd[0007:imb...] Sts[0280:.....] Device ISA bridge
0d:1 8086:7010.00 Cmd[0005:i.b...] Sts[0280:.....] Device IDE
controller
0e:0 1011:0021.02 Cmd[0107:imb..s] Sts[0280:.....] PciBridge 0->1-1
PCI-PCI bridge
10:0 102b:0519.01 Cmd[0083:im....] Sts[0280:.....] Device VGA
compatible controller
PCI Bus 1
08:0 10b7:9050.00 Cmd[0107:imb..s] Sts[0200:.....] Device Ethernet
09:0 9004:8178.00 Cmd[0117:imb..s] Sts[0280:.....] Device SCSI
controller
```

This example displays verbose information about the devices on the primary bus. The two-digit number at the beginning of each line is the device number; the one-digit number following it is the function number:

```
dbgcmd

kd> !pci 1 0
PCI Bus 0
00:0 8086:1237.02 Cmd[0106:.mb..s] Sts[2280:.....] Device Host bridge
cf8:80000000 IntPin:0 IntLine:0 Rom:0 cis:0 cap:0

0d:0 8086:7000.01 Cmd[0007:imb...] Sts[0280:.....] Device ISA bridge
cf8:80006800 IntPin:0 IntLine:0 Rom:0 cis:0 cap:0
```

```
0d:1 8086:7010.00 Cmd[0005:i.b...] Sts[0280:.....] Device IDE
controller
    cf8:80006900 IntPin:0 IntLine:0 Rom:0 cis:0 cap:0
    IO[4]:ffff1

0e:0 1011:0021.02 Cmd[0107:imb..s] Sts[0280:.....] PciBridge 0->1-1
PCI-PCI bridge
    cf8:80007000 IntPin:0 IntLine:0 Rom:0 cap:0 2sts:2280 BCtrl:6
ISA
    IO:f000-ffff Mem:fc000000-fdfffff PMem:fff00000-fffff

10:0 102b:0519.01 Cmd[0083:im....] Sts[0280:.....] Device VGA
compatible controller
    cf8:80008000 IntPin:1 IntLine:9 Rom:80000000 cis:0 cap:0
    MEM[0]:fe800000 MPF[1]:fe000008
```

This example shows even more detailed information about bus 0 (zero), device 0x0D, and function 0x1, including the raw DWORDS from addresses between 0x00 and 0x3F:

```
dbgcmd

kd> !pci f 0 d 1 0 3f
PCI Bus 0
0d:1 8086:7010.00 Cmd[0005:i.b...] Sts[0280:.....] Device IDE
controller
    cf8:80006900 IntPin:0 IntLine:0 Rom:0 cis:0 cap:0
    IO[4]:ffff1
    00000000: 70108086 02800005 01018000 00002000
    00000010: 00000000 00000000 00000000 00000000
    00000020: 0000fff1 00000000 00000000 00000000
    00000030: 00000000 00000000 00000000 00000000
```

This example displays the configuration space for segment 1, bus 0, device 1:

```
dbgcmd

0: kd> !pci 301 1 0 1

PCI Configuration Space (Segment:0001 Bus:00 Device:01 Function:00)
Common Header:
    00: VendorID      14e4 Broadcom Corporation
    02: DeviceID      16c7
    04: Command        0146 MemSpaceEn BusInitiate PERREn SERREn
    06: Status         02b0 CapList 66MHzCapable FB2BCapable DEVSELTiming:1
    .
    .
    .
    5a: MsgCtrl        64BitCapable MultipleMsgEnable:0 (0x1)
MultipleMsgCapable:3 (0x8)
    5c: MsgAddr        2d4bf00
```

```
60: MsgAddrHi      1ae09097  
64: MsData         9891
```

To display all devices and buses on valid segments, issue the command **!pci 602 ffff ff**:

```
dbgcmd  
  
0: kd> !pci 602 ffff ff  
Scanning the following PCI segments: 0 0x1  
PCI Segment 0 Bus 0  
01:0 14e4:16c7.10 Cmd[0146:.mb.ps] Sts[02b0:c6...] Ethernet Controller  
SubID:103c:1321  
02:0 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller  
SubID:103c:1323  
02:1 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller  
SubID:103c:1323  
03:0 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller  
SubID:103c:1323  
03:1 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller  
SubID:103c:1323  
PCI Segment 0 Bus 0x38  
01:0 14e4:1644.12 Cmd[0146:.mb.ps] Sts[02b0:c6...] Ethernet Controller  
SubID:10b7:1000  
PCI Segment 0 Bus 0x54  
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge  
0x54->0x55-0x55  
PCI Segment 0 Bus 0x70  
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge  
0x70->0x71-0x71  
PCI Segment 0 Bus 0xa9  
01:0 8086:b154.00 Cmd[0147:imb.ps] Sts[0ab0:c6.A.] Intel PCI-PCI Bridge  
0xa9->0xaa-0xaa  
PCI Segment 0 Bus 0xaa  
04:0 1033:0035.41 Cmd[0146:.mb.ps] Sts[0210:c....] NEC USB Controller  
SubID:103c:1293  
04:1 1033:0035.41 Cmd[0146:.mb.ps] Sts[0210:c....] NEC USB Controller  
SubID:103c:aa55  
04:2 1033:00e0.02 Cmd[0146:.mb.ps] Sts[0210:c....] NEC USB2 Controller  
SubID:103c:aa55  
05:0 1002:5159.00 Cmd[0187:imb..s] Sts[0290:c....] ATI VGA Compatible  
Controller SubID:103c:1292  
PCI Segment 0 Bus 0xc6  
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge  
0xc6->0xc7-0xc7  
PCI Segment 0 Bus 0xe3  
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge  
0xe3->0xe4-0xe4  
PCI Segment 0x1 Bus 0  
01:0 14e4:16c7.10 Cmd[0146:.mb.ps] Sts[02b0:c6...] Ethernet Controller  
SubID:103c:1321  
02:0 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller  
SubID:103c:1323  
02:1 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller
```

```
SubID:103c:1323
03:0 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller
SubID:103c:1323
03:1 1000:0030.08 Cmd[0147:imb.ps] Sts[0230:c6...] LSI SCSI Controller
SubID:103c:1323
PCI Segment 0x1 Bus 0x54
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge
0x54->0x55-0x55
PCI Segment 0x1 Bus 0x55
00:0 8086:10b9.06 Cmd[0147:imb.ps] Sts[0010:c....] Intel Ethernet
Controller SubID:8086:1083
PCI Segment 0x1 Bus 0x70
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge
0x70->0x71-0x71
PCI Segment 0x1 Bus 0xc6
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge
0xc6->0xc7-0xc7
PCI Segment 0x1 Bus 0xe3
00:0 103c:403b.00 Cmd[0547:imb.ps] Sts[0010:c....] HP PCI-PCI Bridge
0xe3->0xe4-0xe4
```

!pciir

Article • 04/03/2024

The **!pciir** extension displays the contents of the hardware routing of peripheral component interconnect (PCI) devices to interrupt controller inputs.

```
dbgcmd
```

```
!pciir
```

DLL

Windows XP - Kdexts.dll

Windows Vista and later - Unavailable

Additional Information

This extension command can only be used with an x86-based target computer that does not have the Advanced Configuration and Power Interface (ACPI) enabled.

For similar information on any ACPI-enabled computer, use the **!acpiirqarb** extension.

Feedback

Was this page helpful?

 Yes

 No

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

!pcitree

Article • 10/25/2023

The **!pcitree** extension displays information about PCI device objects, including child PCI buses and CardBus buses, and the devices attached to them.

```
dbgcmd
```

```
!pcitree
```

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about PCI buses and PCI device objects, see the Windows Driver Kit (WDK) documentation.

Remarks

Here is an example:

```
dbgcmd
```

```
kd> !pcitree
```

```
Bus 0x0 (FDO Ext fe517338)
 0600 12378086 (d=0, f=0) devext fe4f4ee8 Bridge/HOST to PCI
 0601 70008086 (d=d, f=0) devext fe4f4ce8 Bridge/PCI to ISA
 0101 70108086 (d=d, f=1) devext fe4f4ae8 Mass Storage Controller/IDE
 0604 00211011 (d=e, f=0) devext fe4f4788 Bridge/PCI to PCI

Bus 0x1 (FDO Ext fe516998)
 0200 905010b7 (d=8, f=0) devext fe515ee8 Network Controller/Ethernet
 0100 81789004 (d=9, f=0) devext fe515ce8 Mass Storage Controller/SCSI
 0300 0519102b (d=10, f=0) devext fe4f4428 Display Controller/VGA

Total PCI Root busses processed = 1
```

To understand this display, consider the final device shown. Its base class is 03, its subclass is 00, its Device ID is 0x0519, and its Vendor ID is 0x102B. These values are all

intrinsic to the device itself.

The number after "d=" is the device number; the number after "f=" is the function number. After "devext" is the device extension address, 0xFE4F4428. Finally, the base class name and the subclass name appear.

To obtain more information about a device, use the [!devext](#) extension command with the device extension address as the argument. For this particular device, the command to use would be:

```
dbgcmd  
kd> !devext fe4f4428 pci
```

If the [!pcitree](#) extension generates an error, this often means that your PCI symbols were not loaded properly. Use [.reload pci.sys](#) to fix this problem.

!pcm

Article • 04/03/2024

The **!pcm** extension displays the specified private cache map. This extension is only available in Windows 2000.

dbgcmd

!pcm Address

Parameters

Address

Specifies the address of the private cache map.

DLL

Windows XP - Kdexts.dll

Windows Vista and later - Unavailable

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the [!cchelp](#) extension reference.

Remarks

This extension is supported only in Windows 2000. In Windows XP and later versions of Windows, use the [dt nt!_PRIVATE_CACHE_MAP Address](#) command.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!pcr

Article • 10/25/2023

The **!pcr** extension displays the current status of the Processor Control Region (PCR) on a specific processor.

```
dbgcmd
```

```
!pcr [Processor]
```

Parameters

Processor

Specifies the processor to retrieve the PCR information from. If *Processor* is omitted, the current processor is used.

ⓘ Note

This command is not currently supported and may display incorrect output.

DLL

Kdexts.dll

Additional Information

For information about the PCR and the PRCB, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The processor control block (PRCB) is an extension of the PCR. It can be displayed with the [!prcb](#) extension.

Here is an example of the **!pcr** extension on an x86 target computer:

```
dbgcmd
```

```
kd> !pcr 0
KPCR for Processor 0 at ffdff000:
Major 1 Minor 1
  NtTib.ExceptionList: 801626e0
    NtTib.StackBase: 801628f0
    NtTib.StackLimit: 8015fb00
    NtTib.SubSystemTib: 00000000
    NtTib.Version: 00000000
    NtTib.UserPointer: 00000000
    NtTib.SelfTib: 00000000

      SelfPcr: ffdff000
      Prcb: ffdff120
      Irql: 00000000
      IRR: 00000000
      IDR: ffffffff
      InterruptMode: 00000000
      IDT: 80043400
      GDT: 80043000
      TSS: 803cc000

  CurrentThread: 8015e8a0
  NextThread: 00000000
  IdleThread: 8015e8a0

DpcQueue: 0x80168ee0 0x80100d04 ntoskrnl!KiTimerExpiration
```

One of the entries in this display shows the interrupt request level (IRQL). The `!pcr` extension shows the current IRQL, but the current IRQL is usually not of much interest. The IRQL that existed just before the bug check or debugger connection is more interesting. This is displayed by `!irql`, which is only available on computers running Windows Server 2003 or later versions of Windows.

!pcrs

Article • 04/03/2024

The **!pcrs** extension displays the Intel Itanium-specific processor control registers.

```
dbgcmd
```

```
!pcrs Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address

Specifies the address of a processor control registers file.

DLL

Kdexts.dll

This extension command can only be used with an Itanium-based target computer.

Remarks

Do not confuse the **!pcrs** extension with the **!pcr** extension, which displays the current status of the processor control region.

Feedback

Was this page helpful?

 Yes

 No

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

!pfn

Article • 10/25/2023

The **!pfn** extension displays information about a specific page frame or the entire page frame database.

```
dbgcmd
```

```
!pfn PageFrame
```

Parameters

PageFrame

Specifies the hexadecimal number of the page frame to be displayed.

DLL

Kdexts.dll

Additional Information

For information about page tables, page directories, and page frames, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The page frame number for a virtual address can be obtained by using the [!pte](#) extension.

Here is an example of the output from this extension:

```
dbgcmd
```

```
kd> !pte 801544f4
801544F4 - PDE at C0300800      PTE at C0200550
            contains 0003B163      contains 00154121
            pfn 3b G-DA--KNV    pfn 154 G--A--KRV

kd> !pfn 3b
PFN 0000003B at address 82350588
flink      00000000  blink / share count 00000221  pteaddress C0300800
reference count 0001                      color 0
```

```
restore pte 00000000 containing page      000039 Active

kd> !pfn 154
PFN 00000154 at address 82351FE0
flink      00000000  blink / share count 00000001  pteaddress C0200550
reference count 0001                      color 0
restore pte 00000060 containing page      00003B Active     M
Modified
```

!pmc

Article • 04/03/2024

The **!pmc** extension displays the Performance Monitor Counter (PMC) register at the specified address.

This extension is supported only on an Itanium-based target computer.

dbgcmd

```
!pmc [- Option] Expression [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Option

Can be any one of the following values:

gen

Displays the register as a generic PMC register.

btb

Displays the register as a branch trace buffer (BTB) PMC register.

Expression

Specifies the hexadecimal address of a PMC. The expressions **@kpfcgen** and **@kpfcbtb** can be used as values for this parameter.

If *Expression* is **@kpfcgen**, the debugger displays the current processor PMC register as a generic PMC register. You can also display the current processor PMC register as a generic PMC register by setting *Option* to **gen** and using **@kpfc4**, **@kpfc5**, **@kpfc6**, or **@kpfc7** for the *Expression* value.

If *Expression* is **@kpfcbtb**, the debugger displays the current processor PMC register as a BTB PMC register. You can also display the current processor PMC register as a BTB PMC register by setting *Option* to **btb** and using **@kpfc12** for the *Expression* value.

DisplayLevel

Can be any one of the following values:

0

Displays only the values of each PMC register field. This is the default.

1

Displays detailed information about the PMC register fields that are not reserved or ignored.

2

Displays detailed information about all PMC register fields, including those that are ignored or reserved.

DLL

Kdexts.dll

Feedback

Was this page helpful?



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

!pmssa

Article • 04/03/2024

The **!pmssa** extension displays a specified processor Minimal State Save Area (also known as Min-StateSave Area).

This extension can only be used with an Itanium-based target computer.

```
dbgcmd
```

```
!pmssa Address
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address

Specifies the address of a processor Min-StateSave Area.

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!powertriage

Article • 04/03/2024

The !powertriage extension displays summary information about the system and device power related components. It also provides links to related commands that can be used to gather additional information. The !powertriage command has no parameters. This command can be used with both live kernel-mode debugging and for crash dump file analysis.

Syntax

```
dbgcmd
!powertriage
```

Parameters

None

DLL

Windows 10 and later - Kdexts.dll

Remarks

The !powertriage extension displays the following information.

1. Power state of the device node along with !podev for all the device objects.
2. Links to [!rcdrkd.rcdrlogdump](#) if the driver has enabled the IFR. For more information about IFR, see [Using Inflight Trace Recorder \(IFR\) in KMDF and UMDF 2 Drivers](#).
3. Links to [!wdfkd.wdfdriverinfo](#) and [!wdfkd.wdflogdump](#) for WDF drivers.
4. Links to !fxdevice for PoFx devices. For more information about PoFX, see [Overview of the Power Management Framework](#). Here is example output from the !powertriage command.

```
dbgcmd
kd> !powertriage
```

System Capabilities :

Machine is not AOAC capable.

Power Capabilities:

PopCapabilities @ 0xfffff8022f6c4380

Misc Supported Features: PwrButton S1 S3 S4 S5 HiberFile FullWake

Processor Features:

Disk Features:

Battery Features:

Wake Caps

Ac OnLine Wake: Sx

Soft Lid Wake: Sx

RTC Wake: S4

Min Device Wake: Sx

Default Wake: Sx

Power Action:

PopAction :fffff8022f6ba550

Current System State...: Working

Target System State....: Unspecified

State.....: - Idle(0)

Devices with allocated Power IRPs:

+ ACPI\PNP0C0C\2&daba3ff&1

0xfffffe00023939ad0 ACPI D0 !podev WAIT_WAKE_IRP !irp Related Threads

+ USB\ROOT_HUB30\5&2c60645a&0&0

0xfffffe0002440ac40 USBXHCI D2 !podev WAIT_WAKE_IRP !irp Related Threads !rcdrlogdump !wdfdriverinfo !wdflogdump

Upper DO 0xfffffe00024415a10 USBHUB3 !podev

+ USB\ROOT_HUB30\5&d91dce5&0&0

0xfffffe00023ed4d30 USBXHCI D2 !podev WAIT_WAKE_IRP !irp Related Threads !rcdrlogdump !wdfdriverinfo !wdflogdump

Upper DO 0xfffffe000249d8040 USBHUB3 !podev

+ PCI\VEN_8086&DEV_27E2&SUBSYS_01DE1028&REV_01\3&172e68dd&0&E5

0xfffffe000239e5880 pci D0 !podev FxDevice: !fxdevice WAIT_WAKE_IRP !irp Related Threads

Upper DO 0xfffffe000239c0e50 ACPI !podev

Upper DO 0xfffffe000239f7040 pci !podev

+ PCI\VEN_14E4&DEV_167A&SUBSYS_01DE1028&REV_02\4&24ac2e11&0&00E5

0xfffffe000231e6060 pci D0 !podev WAIT_WAKE_IRP !irp Related Threads

Upper DO 0xfffffe00024359050 b57nd60a !podev

Device Tree Info:

```
!devpowerstate  
!devpowerstate Complete
```

Links:
!poaction
!cstriage
!pdctriage
!pdcclients
!fxdevice
!pnptriage

Dump File Power Failure Analysis

The !powertriage extension can be useful in examining system crashes related to incorrect power state information. For example, in the case of [Bug Check 0x9F: DRIVER_POWER_STATE_FAILURE](#), the extension will display all the allocated power IRPs, the associated device stacks along with:

1. Links to the [!irp](#) command for the related IRPs.
2. Links to the [!findthreads](#) command with the related IRP. The IRP is added as part of the search criteria and displays the threads starting with higher correlation to the search criteria threads listed first. Dumping all device stacks with power IRPs can help in debugging cases where !analyze has not been able to correctly identify the IRP associated with the crash.

Feedback

Was this page helpful?

 Yes

 No

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

!pnpevent

Article • 04/03/2024

The **!pnpevent** extension displays the Plug and Play device event queue.

```
dbgcmd
```

```
!pnpevent [DeviceEvent]
```

Parameters

DeviceEvent

Specifies the address of a device event to display. If this is zero or omitted, the tree of all device events in the queue is displayed.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about Plug and Play drivers, see the Windows Driver Kit (WDK) documentation.

See also

[Plug and Play and Power Debugger Commands](#)

Feedback

Was this page helpful?

 Yes

 No

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

!pocaps

Article • 10/25/2023

The **!pocaps** extension displays the power capabilities of the target computer.

```
dbgcmd
```

```
!pocaps
```

DLL

Kdexts.dll

Additional Information

To view the system's power policy, use the [!popolicy](#) extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of this command's output:

```
dbgcmd
```

```
kd> !pocaps
PopCapabilities @ 0x8016b100
  Misc Supported Features: S4 FullWake
  Processor Features:
    Disk Features:           SpinDown
    Battery Features:
    Wake Caps
      Ac OnLine Wake:       Sx
      Soft Lid Wake:        Sx
      RTC Wake:              Sx
      Min Device Wake:      Sx
      Default Wake:          Sx
```

!pool

Article • 10/25/2023

The **!pool** extension displays information about a specific pool allocation or about the entire system-wide pool.

dbgcmd

```
!pool [Address [Flags]]
```

Parameters

Address

Specifies the pool entry to be displayed. If *Address* is -1, this command displays information about all heaps in the process.

If *Address* is 0 or omitted, this command displays information about the process heap.

Flags

Specifies the level of detail to be used. This can be any combination of the following bit values; the default is zero:

Bit 0 (0x1)

Causes the display to include the pool contents, not just the pool headers.

Bit 1 (0x2)

Causes the display to suppress pool header information for all pools, except the one that actually contains the specified *Address*.

Bit 31 (0x80000000)

Suppresses the description of the pool type and pool tag in the display.

DLL

Kdexts.dll

Additional Information

For information about memory pools, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

In Windows XP and later versions of Windows, the **!pool** extension displays the pool tag associated with each allocation. The owner of that pool tag is also displayed. This display is based on the contents of the pooltag.txt file. This file is located in the triage subdirectory of your Debugging Tools for Windows installation. If you want , you can edit this file to add additional pool tags relevant to your project.

Warning If you install an updated version of Debugging Tools for Windows in the same directory as the current version, it overwrites all of the files in that directory, including pooltag.txt. If you modify or replace the sample pooltag.txt file, be sure to save a copy of it to a different directory. After reinstalling the debuggers, you can copy the saved pooltag.txt over the default version.

If the **!pool** extension reports pool corruption, you should use **!poolval** to investigate.

Here is an example. If *Address* specifies 0xE1001050, the headers of all pools in this block are displayed, and 0xE1001050 itself is marked with an asterisk (*).

```
dbgcmd

kd> !pool e1001050
e1001000 size: 40 previous size: 0 (Allocated) MmDT
e1001040 size: 10 previous size: 40 (Free) Mm
*e1001050 size: 10 previous size: 10 (Allocated) *ObDi
e1001060 size: 10 previous size: 10 (Allocated) ObDi
e1001070 size: 10 previous size: 10 (Allocated) Symt
e1001080 size: 40 previous size: 10 (Allocated) ObDm
e10010c0 size: 10 previous size: 40 (Allocated) ObDi
....
```

In this example, the right-most column shows the pool tag. The column to the left of this shows whether the pool is free or allocated.

The following command shows the pool headers and pool contents:

```
dbgcmd

kd> !pool e1001050 1
e1001000 size: 40 previous size: 0 (Allocated) MmDT
e1001008 ffffffff 0057005c 004e0049 004f0044
e1001018 ffffffff 0053005c 00730079 00650074

e1001040 size: 10 previous size: 40 (Free) Mm
e1001048 ffffffff e1007ba8 e1501a58 01028101
e1001058 ffffffff 00000000 e1000240 01028101
```

```
*e1001050 size: 10 previous size: 10 (Allocated) *ObDi
e1001058 ffffffff 00000000 e1000240 01028101
e1001068 ffffffff 00000000 e10009c0 01028101

e1001060 size: 10 previous size: 10 (Allocated) ObDi
e1001068 ffffffff 00000000 e10009c0 01028101
e1001078 ffffffff 00000000 00000000 04028101

.....
```

!poolfind

Article • 10/25/2023

The **!poolfind** extension finds all instances of a specific pool tag in either nonpaged or paged memory pools.

dbgcmd

```
!poolfind TagString [PoolType]  
!poolfind TagValue [PoolType]
```

Parameters

TagString

Specifies the pool tag. *TagString* is a case-sensitive ASCII string. The asterisk (*) can be used to represent any number of characters; the question mark (?) can be used to represent exactly one character. Unless an asterisk is used, *TagString* must be exactly four characters in length.

TagValue

Specifies the pool tag. *TagValue* must begin with "0x", even if the default radix is 16. If this parameter begins with any other value (including "0X") it will be interpreted as an ASCII tag string.

PoolType

Specifies the type of pool to be searched. The following values are permitted:

0

Specifies nonpaged memory pool. This is the default.

1

Specifies paged memory pool.

2

Specifies the special pool.

4

Specifies the session pool.

DLL

Kdexts.dll

Additional Information

For information about memory pools and pool tags, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This command can take a significant amount of time to execute, depending on the size of pool memory that must be searched. To speed up this execution, increase the COM port speed with the [CTRL+A \(Toggle Baud Rate\)](#) key, or use the [.cache \(Set Cache Size\)](#) command to increase the cache size (to approximately 10 MB).

The pool tag is the same tag passed to the `ExAllocateXxx` family of routines.

Here is an example. The entire nonpaged pool is searched and then the paged pool is searched, but the command is terminated before completion (after an hour of operation):

```
dbgcmd

kd> !poolfind SeSd 0

Scanning large pool allocation table for Tag: SeSd (827d1000 : 827e9000)

Searching NonPaged pool (823b1000 : 82800000) for Tag: SeSd

826fa130 size: c0 previous size: 40 (Allocated) SeSd
82712000 size: c0 previous size: 0 (Allocated) SeSd
82715940 size: a0 previous size: 60 (Allocated) SeSd
8271da30 size: c0 previous size: 10 (Allocated) SeSd
82721c00 size: 10 previous size: 30 (Free) SeSd
8272b3f0 size: 60 previous size: 30 (Allocated) SeSd
8272d770 size: 60 previous size: 40 (Allocated) SeSd
8272d7d0 size: a0 previous size: 60 (Allocated) SeSd
8272d960 size: a0 previous size: 70 (Allocated) SeSd
82736f30 size: a0 previous size: 10 (Allocated) SeSd
82763840 size: a0 previous size: 10 (Allocated) SeSd
8278b730 size: 100 previous size: 290 (Allocated) SeSd
8278b830 size: 10 previous size: 100 (Free) SeSd
82790130 size: a0 previous size: 20 (Allocated) SeSd
82799180 size: a0 previous size: 10 (Allocated) SeSd
827c00e0 size: a0 previous size: 30 (Allocated) SeSd
827c8320 size: a0 previous size: 60 (Allocated) SeSd
827ca180 size: a0 previous size: 50 (Allocated) SeSd
827ec140 size: a0 previous size: 10 (Allocated) SeSd

Searching NonPaged pool (fe7c3000 : ffbe0000) for Tag: SeSd
```

```
kd> !poolfind SeSd 1

Scanning large pool allocation table for Tag: SeSd (827d1000 : 827e9000)

Searching Paged pool (e1000000 : e4400000) for Tag: SeSd

e10000b0 size: d0 previous size: 20 (Allocated) SeSd
e1000260 size: d0 previous size: 60 (Allocated) SeSd
.....
e1221dc0 size: a0 previous size: 60 (Allocated) SeSd
e1224250 size: a0 previous size: 30 (Allocated) SeSd

...terminating - searched pool to e1224000
kd>
```

!poolused

Article • 10/25/2023

The **!poolused** extension displays memory use summaries, based on the tag used for each pool allocation.

dbgcmd

```
!poolused [Flags [TagString]]
```

Parameters

Flags

Specifies the amount of output to be displayed and the method of sorting the output. This can be any combination of the following bit values, except that bits 1 (0x2) and 2 (0x4) cannot be used together. The default is 0x0, which produces summary information, sorted by pool tag.

Bit 0 (0x1)

Displays more detailed (verbose) information.

Bit 1 (0x2)

Sorts the display by the amount of nonpaged memory use.

Bit 2 (0x4)

Sorts the display by the amount of paged memory use.

Bit 3 (0x8)

Displays the session pool instead of the standard pool. You can use the [!session](#) command to switch between sessions.

TagString

Specifies the pool tag. *TagString* is a case-sensitive ASCII string. The asterisk (*) can be used to represent any number of characters; the question mark (?) can be used to represent exactly one character. Unless an asterisk is used, *TagString* must be exactly four characters in length.

DLL

Kdexts.dll

Additional Information

For information about memory pools and pool tags, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!poolused** extension gathers data from the pool tagging feature of Windows. Pool tagging is permanently enabled on Windows Server 2003 and later versions of Windows.

If you stop executing the extension before it completes, the debugger displays partial results.

The display for this command shows the memory use for each tag in paged pool and nonpaged pool. In both cases, the display includes the number of currently outstanding allocations for the given tag and the number of bytes being consumed by those allocations.

Here is a partial example of the output from this extension:

```
dbgcmd

0: kd> !poolused
      Sorting by Tag

      Pool Used:
      NonPaged          Paged
      Tag    Allocs   Used    Allocs   Used
      1394        1     520        0     UNKNOWN pooltag '1394', please
      update pooltag.txt
      1MEM        1    3368        0     UNKNOWN pooltag '1MEM', please
      update pooltag.txt
      2MEM        1    3944        0     UNKNOWN pooltag '2MEM', please
      update pooltag.txt
      3MEM        3     248        0     UNKNOWN pooltag '3MEM', please
      update pooltag.txt
      8042        4    3944        0     OPS/2 kb and mouse , Binary:
      i8042prt.sys
      AGP         1     344        2     384UNKNOWN pooltag 'AGP ', please
      update pooltag.txt
      AcdN        2    1072        0     0TDI AcdObjectInfoG
      AcpA        3     192        1     504ACPI Pooltags , Binary: acpi.sys
      AcpB        0      0        4     576ACPI Pooltags , Binary: acpi.sys
```

AcpD	40	13280	0	0ACPI Pooltags , Binary: acpi.sys
AcpF	6	240	0	0ACPI Pooltags , Binary: acpi.sys
AcpM	0	0	1	128ACPI Pooltags , Binary: acpi.sys
AcpO	4	208	0	0ACPI Pooltags , Binary: acpi.sys
 ...				
WmiG	30	6960	0	0Allocation of WMIGUID
WmiR	63	4032	0	0Wmi Registration info blocks
Wmip	146	3504	182	18600Wmi General purpose allocation
Wmit	1	4096	7	49480Wmi Trace
Wrpa	2	720	0	0WAN_ADAPTER_TAG
Wrpc	1	72	0	0WAN_CONN_TAG
Wrpi	1	120	0	0WAN_INTERFACE_TAG
Wrps	2	128	0	0WAN_STRING_TAG
aEoP	1	672	0	0UNKNOWN pooltag 'aEoP', please
update pooltag.txt				
fEoP	1	16	0	0UNKNOWN pooltag 'fEoP', please
update pooltag.txt				
hSVD	0	0	1	40Shared Heap Tag , Binary:
mrxdav.sys				
hibr	0	0	1	24576UNKNOWN pooltag 'hibr', please
update pooltag.txt				
iEoP	1	24	0	0UNKNOWN pooltag 'iEoP', please
update pooltag.txt				
idle	2	208	0	0Power Manager idle handler
jEoP	1	24	0	0UNKNOWN pooltag 'jEoP', please
update pooltag.txt				
mEoP	1	88	0	0UNKNOWN pooltag 'mEoP', please
update pooltag.txt				
ohci	1	136	0	01394 OHCI host controller driver
rx..	3	1248	0	0UNKNOWN pooltag ' rx', please
update pooltag.txt				
sidg	2	48	0	0GDI spooler events
thdd	0	0	1	20480DirectDraw/3D handle manager table
usbp	18	77056	2	96UNKNOWN pooltag 'usbp', please
update pooltag.txt				
vPrt	0	0	18	68160UNKNOWN pooltag 'vPrt', please
update pooltag.txt				
TOTAL	3570214	209120008	38769	13066104

!poolval

Article • 04/03/2024

The **!poolval** extension analyzes the headers for a pool page and diagnoses any possible corruption.

dbgcmd

```
!poolval Address [DisplayLevel]
```

Parameters

Address

Specifies the address of the pool whose header is to be analyzed.

DisplayLevel

Specifies the information to include in the display. This can be any of the following values (the default is zero):

0

Causes basic information to be displayed.

1

Causes basic information and linked header lists to be displayed.

2

Causes basic information, linked header lists, and basic header information to be displayed.

3

Causes basic information, linked header lists, and full header information to be displayed.

DLL

Kdexts.dll

Additional Information

For information about memory pools, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Feedback

Was this page helpful?

 Yes

 No

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

!popolicy

Article • 04/03/2024

The **!popolicy** extension displays the power policy of the target computer.

```
dbgcmd
```

```
!popolicy [Address]
```

Parameters

Address

Specifies the address of the power policy structure to display. If this is omitted, then `nt!PopPolicy` is displayed.

DLL

Kdexts.dll

Additional Information

To view the system's power capabilities, use the [!pocaps](#) extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of this command's output:

```
dbgcmd
```

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80164d58
  PowerButton:      Shutdown  Flags: 00000003  Event: 00000000  Query UI
  SleepButton:      None     Flags: 00000003  Event: 00000000  Query UI
  LidClose:         None     Flags: 00000001  Event: 00000000  Query
  Idle:             None     Flags: 00000001  Event: 00000000  Query
  OverThrottled:    None     Flags: c0000004  Event: 00000000  Override
  Nowakes Critical
  IdleTimeout:      0  IdleSensitivity:        50%
```

MinSleep:	S0	MaxSleep:	S0
LidOpenWake:	S0	FastSleep:	S0
WinLogonFlags:	1	S4Timeout:	0
VideoTimeout:	0	VideoDim:	209
SpinTimeout:	0	OptForPower:	1
FanTolerance:	0%	ForcedThrottle:	0%
MinThrottle:	0%		

See also

[Plug and Play and Power Debugger Commands](#)

Feedback

Was this page helpful?

 Yes

 No

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

!pplookaside

Article • 04/03/2024

The **!pplookaside** extension command displays [Lookaside Lists](#) for processors in the target computer.

dbgcmd

```
!pplookaside <address>
```

Parameters

ParamName

The address of the processor.

DLL

kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppmidle

Article • 04/03/2024

The **!ppmidle** extension command displays processor idle states.

```
dbgcmd
```

```
!ppmidle
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7
- Windows 8
- Windows 8.1
- Windows 10, Version 1511
- Windows 10, Version 1607
- Windows 10, Version 1703

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppmidleaccounting

Article • 04/03/2024

The **!ppmidleaccounting** extension command displays processor idle state accounting information.

```
dbgcmd
```

```
!ppidleaccounting
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7
- Windows 8
- Windows 8.1
- Windows 10, Version 1511
- Windows 10, Version 1607
- Windows 10, Version 1703

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppmidlepolicy

Article • 04/03/2024

The `!ppmidlepolicy` extension command displays C-state policy.

```
dbgcmd
```

```
!ppmidlepolicy
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 7

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppm!pscenarios

Article • 04/03/2024

The **!ppm!pscenarios** extension command displays per-scenario policy overrides.

```
dbgcmd
```

```
!ppm!pscenarios
```

Parameters

None

This command is supported on the following versions of Windows:

- Windows 8

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppmperf

Article • 04/03/2024

The **!ppmperf** extension command displays power and performance constraint information.

```
dbgcmd
```

```
!ppmperf
```

Parameters

None

DLL

kedexts.dll

NOTE: This command is not supported on all versions of Windows, such as Windows 10, Version 1703.

Feedback

Was this page helpful?

 Yes

 No

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

!ppmperfpolicy

Article • 04/03/2024

The `!ppmperfpolicy` extension command is no longer supported, use the [!ppmsettings](#) command instead.

dbgcmd

`!ppmperfpolicy`

Parameters

None

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

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

!ppmsettings

Article • 04/03/2024

The `!ppmsettings` extension command displays currently active ppm settings for the processor.

```
dbgcmd
!ppmsettings
```

Parameters

None

Remarks

This command is supported on the following versions of Windows:

- Windows 10, Version 1511
- Windows 10, Version 1607
- Windows 10, Version 1703

DLL

Kdexts.dll

Feedback

Was this page helpful?



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

!ppmstate

Article • 04/03/2024

The **!ppmstate** extension command displays power state information.

```
dbgcmd
```

```
!ppmstate
```

Parameters

None

DLL

Kdexts.dll

NOTE: This command is not supported on all versions of Windows, such as Windows 10, Version 1703.

Feedback

Was this page helpful?

 Yes

 No

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

!prcb

Article • 04/03/2024

The **!prcb** extension displays the processor control block (PRCB).

```
dbgcmd
```

```
!prcb [Processor]
```

Parameters

Processor

Specifies the processor to retrieve the PRCB information from. If *Processor* is omitted, processor zero will be used.

DLL

Kdexts.dll

Additional Information

For information about the PCR and the PRCB, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The PRCB is an extension of the processor control region (PCR). To display the PCR, use the [!pcr](#) extension.

Here is an example:

```
dbgcmd
```

```
kd> !prcb
PRCB for Processor 0 at e0000000818ba000:
Threads-- Current e0000000818bbe10 Next 0000000000000000 Idle
e0000000818bbe10
Number 0 SetMember 00000001
Interrupt Count -- 0000b81f
```

```
Times -- Dpc      00000028 Interrupt 000003ff
Kernel 00005ef4 User          00000385
```

Feedback

Was this page helpful?

 Yes

 No

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

!process

Article • 10/25/2023

The !process extension displays information about the specified process, or about all processes, including the EPROCESS block.

This extension can be used only during kernel-mode debugging.

Syntax

dbgcmd

```
!process [/s Session] [/m Module] [Process [Flags]]  
!process [/s Session] [/m Module] 0 Flags ImageName
```

Parameters

/s **** *Session*

Specifies the session that owns the desired process.

/m **** *Module*

Specifies the module that owns the desired process.

Process

Specifies the hexadecimal address or the process ID of the process on the target computer.

The value of *Process* determines whether the !process extension displays a process address or a process ID . If *Process* is omitted in any version of Windows, the debugger displays data only about the current system process. If *Process* is 0 and *ImageName* is omitted, the debugger displays information about all active processes. If -1 is specified for *Process* information about the current process is displayed.

Flags

Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default varies according to the version of Windows and the value of *Process*. The default is 0x3 if *Process* is omitted or if *Process* is either 0 or -1; otherwise, the default is 0xF.

Bit 0 (0x1)

Displays time and priority statistics.

Bit 1 (0x2)

Displays a list of threads and events associated with the process, and their wait states.

Bit 2 (0x4)

Displays a list of threads associated with the process. If this is included without Bit 1 (0x2), each thread is displayed on a single line. If this is included along with Bit 1, each thread is displayed with a stack trace.

Bit 3 (0x8)

Displays the return address and the stack pointer for each function. The display of function arguments is suppressed.

Bit 4 (0x10)

Sets the process context equal to the specified process for the duration of this command. This results in a more accurate display of thread stacks. Because this flag is equivalent to using [.process /p /r](#) for the specified process, any existing user-mode module list will be discarded. If *Process* is zero, the debugger displays all processes, and the process context is changed for each one. If you are only displaying a single process and its user-mode state has already been refreshed (for example, with [.process /p /r](#)), it is not necessary to use this flag. This flag is only effective when used with Bit 0 (0x1).

ImageName

Specifies the name of the process to be displayed. The debugger displays all processes whose executable image names match *ImageName*. The image name must match that in the EPROCESS block. In general, this is the executable name that was invoked to start the process, including the file extension (usually .exe), and truncated after the fifteenth character. There is no way to specify an image name that contains a space. When *ImageName* is specified, *Process* must be zero.

DLL

Kdexts.dll

Additional Information

For information about processes in kernel mode, see [Changing Contexts](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The following is an example of a **!process 0 0** display:

dbgcmd

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 80a02a60 Cid: 0002 Peb: 00000000 ParentCid: 0000
    DirBase: 00006e05 ObjectTable: 80a03788 TableSize: 150.
    Image: System
PROCESS 80986f40 Cid: 0012 Peb: 7ffde000 ParentCid: 0002
    DirBase: 000bd605 ObjectTable: 8098fce8 TableSize: 38.
    Image: smss.exe
PROCESS 80958020 Cid: 001a Peb: 7ffde000 ParentCid: 0012
    DirBase: 0008b205 ObjectTable: 809782a8 TableSize: 150.
    Image: csrss.exe
PROCESS 80955040 Cid: 0020 Peb: 7ffde000 ParentCid: 0012
    DirBase: 00112005 ObjectTable: 80955ce8 TableSize: 54.
    Image: winlogon.exe
PROCESS 8094fce0 Cid: 0026 Peb: 7ffde000 ParentCid: 0020
    DirBase: 00055005 ObjectTable: 80950cc8 TableSize: 222.
    Image: services.exe
PROCESS 8094c020 Cid: 0029 Peb: 7ffde000 ParentCid: 0020
    DirBase: 000c4605 ObjectTable: 80990fe8 TableSize: 110.
    Image: lsass.exe
PROCESS 809258e0 Cid: 0044 Peb: 7ffde000 ParentCid: 0026
    DirBase: 001e5405 ObjectTable: 80925c68 TableSize: 70.
    Image: SPOOLSS.EXE
```

The following table describes some of the elements of the **!process 0 0** output.

Element	Meaning
Process address	The eight-character hexadecimal number after the word PROCESS is the address of the EPROCESS block. In the final entry in the preceding example, the process address is 0x809258E0.
Process ID (PID)	The hexadecimal number after the word Cid. In the final entry in the preceding example, the PID is 0x44, or decimal 68.
Process Environment Block (PEB)	The hexadecimal number after the word Peb is the address of the process environment block. In the final entry in the preceding example, the PEB is located at address 0x7FFDE000.
Parent process PID	The hexadecimal number after the word ParentCid is the PID of the parent process. In the final entry in the preceding example, the parent process PID is 0x26, or decimal 38.

Element	Meaning
Image	The name of the module that owns the process. In the final entry in the preceding example, the owner is spoolss.exe. In the first entry, the owner is the operating system itself.
Process object address	The hexadecimal number after the word ObjectTable. In the final entry in the preceding example, the address of the process object is 0x80925c68.

To display full details on one process, set *Flags* to 7. The process itself can be specified by setting *Process* equal to the process address, setting *Process* equal to the process ID, or setting *ImageName* equal to the executable image name. Here is an example:

dbgcmd

```

kd> !process fb667a00 7
PROCESS fb667a00 Cid: 0002 Peb: 00000000 ParentCid: 0000
  DirBase: 00030000 ObjectTable: e1000f88 TableSize: 112.
  Image: System
  VadRoot fb666388 Clone 0 Private 4. Modified 9850. Locked 0.
  FB667BBC MutantState Signalled OwningThread 0
  Token          e10008f0
  ElapsedTime    15:06:36.0338
  UserTime       0:00:00.0000
  KernelTime     0:00:54.0818
  QuotaPoolUsage[PagedPool] 1480
  Working Set Sizes (now,min,max) (3, 50, 345)
    PeakWorkingSetSize 118
    VirtualSize        1 Mb
    PeakVirtualSize    1 Mb
    PageFaultCount    992
    MemoryPriority     BACKGROUND
    BasePriority       8
    CommitCharge       8

  THREAD fb667780 Cid 2.1 Peb: 00000000 Win32Thread: 80144900 WAIT:
(WrFreePage) KernelMode Non-Alertable
  80144fc0 SynchronizationEvent
  Not impersonating
  Owning Process fb667a00
  WaitTime (seconds) 32278
  Context Switch Count 787
  UserTime       0:00:00.0000
  KernelTime     0:00:21.0821
  Start Address Phase1Initialization (0x801aab44)
  Initial Sp fb26f000 Current Sp fb26ed00
  Priority 0 BasePriority 0 PriorityDecrement 0 DecrementCount 0

  ChildEBP RetAddr Args to Child

```

```
fb26ed18 80118efc c0502000 804044b0 00000000 KiSwapThread+0xb5
fb26ed3c 801289d9 80144fc0 00000008 00000000 KeWaitForSingleObject+0x1c2
```

Note that the address of the process object can be used as input to other extensions, such as [!Handle](#), to obtain further information.

The following table describes some of the elements in the previous example.

Element	Meaning
WAIT	The parenthetical comment after this heading gives the reason for the wait. The command dt nt!_KWAIT_REASON will display a list of all wait reasons.
Elapsed Time	Lists the amount of time that has elapsed since the process was created. This is displayed in units of Hours:Minutes:Seconds.Milliseconds.
User Time	Lists the amount of time the process has been running in user mode. If the value for UserTime is exceptionally high, it might identify a process that is depleting system resources. Units are the same as those of ElapsedTime.
Kernel Time	Lists the amount of time the process has been running in kernel mode. If the value for KernelTime is exceptionally high, it might identify a process that is depleting system resources. Units are the same as those of ElapsedTime.
Working Set sizes	Lists the current, minimum and maximum working set size for the process, in pages. An exceptionally large working set size can be a sign of a process that is leaking memory or depleting system resources.
QuotaPoolUsage entries	Lists the paged and nonpaged pool used by the process. On a system with a memory leak, looking for excessive nonpaged pool usage on all the processes can tell you which process has the memory leak.
Clone	Indicates whether or not the process was created by the POSIX or Interix subsystems.
Private	Indicates the number of private (non-sharable) pages currently being used by the process. This includes both paged in and paged out memory.

In addition to the process list information, the thread information contains a list of the resources on which the thread has locks. This information is listed in the third line of output after the thread header. In this example, the thread has a lock on one resource, a SynchronizationEvent with an address of 80144fc0. By comparing this address to the list of locks shown by the **!kdext*.locks** extension, you can determine which threads have exclusive locks on resources.

The **!stacks** extension gives a brief summary of the state of every thread. This can be used instead of the **!process** extension to get a quick overview of the system, especially when debugging multithread issues, such as resource conflicts or deadlocks.

!processfields

Article • 10/25/2023

The **!processfields** extension displays the names and offsets of the fields within the executive process (EPROCESS) block.

```
dbgcmd
```

```
!processfields
```

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable (see the Remarks section)

Additional Information

For information about the EPROCESS block, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension command is not available in Windows XP or later versions of Windows. Instead, use the [dt \(Display Type\)](#) command to show the EPROCESS structure directly:

```
dbgcmd
```

```
kd> dt nt!_EPROCESS
```

Here is an example of **!processfields** from a Windows 2000 system:

```
dbgcmd
```

```
kd> !processfields
EPROCESS structure offsets:
```

Pcb:	0x0
ExitStatus:	0x6c
LockEvent:	0x70
LockCount:	0x80
CreateTime:	0x88

ExitTime:	0x90
LockOwner:	0x98
UniqueProcessId:	0x9c
ActiveProcessLinks:	0xa0
QuotaPeakPoolUsage[0]:	0xa8
QuotaPoolUsage[0]:	0xb0
PagefileUsage:	0xb8
CommitCharge:	0xbc
PeakPagefileUsage:	0xc0
PeakVirtualSize:	0xc4
VirtualSize:	0xc8
Vm:	0xd0
DebugPort:	0x120
ExceptionPort:	0x124
ObjectTable:	0x128
Token:	0x12c
WorkingSetLock:	0x130
WorkingSetPage:	0x150
ProcessOutswapEnabled:	0x154
ProcessOutswapped:	0x155
AddressSpaceInitialized:	0x156
AddressSpaceDeleted:	0x157
AddressCreationLock:	0x158
ForkInProgress:	0x17c
VmOperation:	0x180
VmOperationEvent:	0x184
PageDirectoryPte:	0x1f0
LastFaultCount:	0x18c
VadRoot:	0x194
VadHint:	0x198
CloneRoot:	0x19c
NumberOfPrivatePages:	0x1a0
NumberOfLockedPages:	0x1a4
ForkWasSuccessful:	0x182
ExitProcessCalled:	0x1aa
CreateProcessReported:	0x1ab
SectionHandle:	0x1ac
Peb:	0x1b0
SectionBaseAddress:	0x1b4
QuotaBlock:	0x1b8
LastThreadExitStatus:	0x1bc
WorkingSetWatch:	0x1c0
InheritedFromUniqueProcessId:	0x1c8
GrantedAccess:	0x1cc
DefaultHardErrorProcessing	0x1d0
LdtInformation:	0x1d4
VadFreeHint:	0x1d8
VdmObjects:	0x1dc
DeviceMap:	0x1e0
ImageFileName[0]:	0x1fc
VmTrimFaultValue:	0x20c
Win32Process:	0x214
Win32WindowStation:	0x1c4

!processirps

Article • 10/25/2023

The **!processirps** extension displays information about I/O request packets (IRPs) associated with processes.

```
dbgcmd  
  !processirps  
  !processirps ProcessAddress [Flags]
```

Parameters

**** *ProcessAddress*

The address of a process. If you specify *ProcessAddress*, only IRPs associated with that process are displayed. If you do not specify *ProcessAddress*, IRPs for all processes are displayed.

**** *Flags*

A bitwise OR of one or more of the following flags.

Bit 0 (0x1)

Display IRPs queued to threads.

Bit 1 (0x2)

Display IRPs queued to file objects.

If you specify *Flags*, you must also specify *ProcessAddress*. If you do not specify *Flags*, IRPs queued to both threads and file objects are displayed.

DLL

kdexts.dll

Remarks

This command enables you to quickly identify any queued IRPs for a process, both those that are queued to threads and those that are queued to file objects. IRPs are queued to a file object when the file object has a completion port associated with it.

Examples

You can use `!process` command to get process addresses. For example, you could get the process address for explorer.exe.

```
dbgcmd

2: kd> !process 0 0
***** NT ACTIVE PROCESS DUMP *****
...
PROCESS ffffffa800688c940
SessionId: 1 Cid: 0bbc Peb: 7f70da5e000 ParentCid: 0b84
DirBase: 2db10000 ObjectTable: fffff8a0025bd440 HandleCount: 1056.
Image: explorer.exe
```

Now you can pass the process address for explorer.exe to the `!processirps` command. The following output shows that explorer.exe has IRPs queued to threads and IRPs queued to file objects.

```
dbgcmd

2: kd> !processirps ffffffa800688c940
***** PROCESS ffffffa800688c940 (Image: explorer.exe) *****

Checking threads for IRPs.

Thread ffffffa800689f080:

    IRP ffffffa80045ccc10 - Owned by \FileSystem\Ntfs for device
ffffffa8004f5c030
    IRP ffffffa800454f650 - Owned by \FileSystem\Ntfs for device
ffffffa8004f5c030
    ...
    IRP ffffffa80068e9c10 - Owned by \FileSystem\Ntfs for device
ffffffa8004f5c030

Checking file objects for IRPs.

FileObject ffffffa80068795e0 (handle 8bc):

    IRP ffffffa8006590cf0 - Owned by \Driver\DeviceApi for device DeviceApi
(ffffffa800363ae40)

    ...
FileObject ffffffa8005bf59c0 (handle 900):

    IRP ffffffa8006659010 - Owned by \Driver\DeviceApi for device DeviceApi
(ffffffa800363ae40)
```

!psp

Article • 04/03/2024

The **!psp** extension displays the processor state parameter (PSP) register at the specified address.

This extension is supported only on Itanium-based target computers.

dbgcmd

```
!psp Address [DisplayLevel]
```

Important This command has been deprecated in the Windows Debugger Version 10.0.14257 and later, and is no longer available.

Parameters

Address

Specifies the hexadecimal address of the PSP register to display.

DisplayLevel

Can be any one of the following options:

0

Displays only the values of each PSP field. This is the default.

1

Displays more in-depth information on each of the PSP fields that is not reserved or ignored.

2

Displays more in-depth information on all of the PSP fields, including those that are ignored or reserved.

DLL

Kdexts.dll

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!pte

Article • 10/25/2023

The **!pte** extension displays the page table entry (PTE) and page directory entry (PDE) for the specified address.

Syntax

```
dbgcmd  
!pte VirtualAddress  
!pte PTE  
!pte LiteralAddress 1
```

Parameters

VirtualAddress

Specifies the virtual address whose page table is desired.

PTE

Specifies the address of an actual PTE.

*LiteralAddress **** 1*

Specifies the address of an actual PTE or PDE.

DLL

Kdexts.dll

Additional Information

For information about page tables, page directories, and an explanation of the status bits, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

If one parameter is supplied and this parameter is an address in the region of memory where the page tables are stored, the debugger treats this as the *PTE* parameter. This parameter is taken as the actual address of the desired PTE, and the debugger will display this PTE and the corresponding PDE.

If one parameter is supplied and this parameter is not an address in this region, the debugger treats this as the *VirtualAddress* parameter. The PTE and PDE that hold the mapping for this address are displayed.

If two parameters are supplied and the second parameter is 1 (or any other small number), the debugger treats the first parameter as *LiteralAddress*. This address is interpreted as the actual address of a PTE and also as the actual address of a PDE, and the corresponding (and possibly invalid) data will be shown.

(x86 or x64 target computer only) If two parameters are supplied and the second parameter is greater than the first, the debugger treats the two parameters as *StartAddress* and *EndAddress*. The command then displays the PTEs for each page in the specified memory range.

For a list of all system PTEs, use the [!sysptes](#) extension.

Here is an example from an x86 target computer:

```
dbgcmd

kd> !pte 801544f4
801544F4 - PDE at C0300800      PTE at C0200550
           contains 0003B163      contains 00154121
           pfn 3b G-DA--KVV     pfn 154 G--A--KRV
```

The first line of this example restates the virtual address being investigated. It then gives the virtual address of the PDE and the PTE, which contain information about the virtual-physical mapping of this address.

The second line gives the actual contents of the PDE and the PTE.

The third line takes these contents and analyzes them, breaking them into the page frame number (PFN) and the status bits.

See the [!pfn](#) extension or the [Converting Virtual Addresses to Physical Addresses](#) section for information about how to interpret and use the PFN.

On an x86 or x64 target computer, the status bits for the PDE and the PTE are shown in the following table. The [!pte](#) display indicates these bits with capital letters or dashes, and adds additional information as well.

Bit	Display when set	Display when clear	Meaning
0x200	C	-	Copy on write.
0x100	G	-	Global.

Bit	Display when set	Display when clear	Meaning
0x80	L	-	Large page. This only occurs in PDEs, never in PTEs.
0x40	D	-	Dirty.
0x20	A	-	Accessed.
0x10	N	-	Cache disabled.
0x8	T	-	Write-through.
0x4	U	K	Owner (user mode or kernel mode).
0x2	W	R	Writeable or read-only. Only on multiprocessor computers and any computer running Windows Vista or later.
0x1	V		Valid.
	E	-	Executable page. For platforms that do not support a hardware execute/noexecute bit, including many x86 systems, the E is always displayed.

!pte2va

Article • 04/03/2024

The **!pte2va** extension displays the virtual address that corresponds to the specified page table entry (PTE).

```
dbgcmd
```

```
!pte2va Address
```

Parameters

Address

Specifies the PTE.

DLL

Kdexts.dll

Additional Information

For information about page tables and PTEs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To examine the contents of a specific PTE, use the [!pte](#) extension.

Here is an example of the output from the **!pte2va** extension:

```
dbgcmd
```

```
kd> !pte2va 9230  
000800000248c000
```

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!ptov

Article • 10/25/2023

The **!ptov** extension displays the entire physical-to-virtual map for a given process.

```
dbgcmd
```

```
!ptov DirBase
```

Parameters

DirBase

Specifies the directory base for the process. To determine the directory base, use the [!process](#) command, and look at the value displayed for DirBase.

DLL

Kdexts.dll

Remarks

Here is an example. First, use [.process](#) and [!process](#) to determine the directory base of the current process:

```
dbgcmd
```

```
1: kd> .process
Implicit process is now 852b4040
1: kd> !process 852b4040 1
PROCESS 852b4040 SessionId: none Cid: 0004 Peb: 00000000 ParentCid:
0000
    DirBase: 00185000 ObjectTable: 83203000 HandleCount: 663.
    Image: System
    ...

```

In this case, the directory base is 0x00185000. Pass this address to **!ptov**:

```
dbgcmd
```

```
1: kd> !ptov 185000
X86PtoV: pagedir 185000, PAE enabled.
15e11000 10000
```

```
549e6000 20000
...
60a000 210000
40b000 211000
...
54ad3000 25f000
548d3000 260000
...
d71000 77510000
...
```

The numbers in the left column are the physical addresses of each memory page that has a mapping for this process. The numbers in the right column are the virtual addresses to which they are mapped.

The total display is very long.

Here is a 64-bit example.

```
dbgcmd

3: kd> .process
Implicit process is now ffffffa80`0361eb30
3: kd> !process ffffffa80`0361eb30 1
PROCESS ffffffa800361eb30
    SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
    DirBase: 00187000 ObjectTable: fffff8a000002870 HandleCount: 919.
    Image: System
    ...
3: kd> !ptov 187000
Amd64PtoV: pagedir 187000
00000001`034fb000 1d0000
a757c000 1d1000
00000001`0103d000 1d2000
c041e000 1d3000
...
2ed6f000 ffffff680`00001000
00000001`13939000 ffffff680`00003000
ceefb000 ffffff680`00008000
...
```

The directory base is the physical address of the first table that is used in virtual address translation. This table has different names depending on the bitness of the target operating system and whether Physical Address Extension (PAE) is enabled for the target operating system.

For 64-bit Windows, the directory base is the physical address of the Page Map Level 4 (PML4) table. For 32-bit Windows with PAE enabled, the directory base is the physical

address of the Page Directory Pointers (PDP) table. For 32-bit Windows with PAE disabled, the directory bas is the physical address of the Page Directory (PD) table.

For related topics, see [!vtop](#) and [Converting Virtual Addresses to Physical Addresses](#). For information about virtual address translation, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

!qllocks

Article • 10/25/2023

The **!qllocks** extension displays the state of all queued spin locks.

```
dbgcmd
```

```
!qllocks
```

DLL

Kdexts.dll

Additional Information

For information about spin locks, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This command is useful only on a multiprocessor system.

Here is an example:

```
dbgcmd
```

```
0: kd> !qllocks
Key: 0 = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt
```

Lock Name	Processor Number			
	0	1	2	3
KE - Dispatcher				
KE - Unused Spare				
MM - PFN				
MM - System Space				
CC - Vacb				
CC - Master				
EX - NonPagedPool				
IO - Cancel				
EX - WorkQueue				
IO - Vpb				
IO - Database				
IO - Completion				
NTFS - Struct				

AFD - WorkQueue

CC - Bcb

MM - MM NonPagedPool

!ready

Article • 10/25/2023

The `!ready` extension displays summary information about each thread in the system in a READY state.

```
dbgcmd
```

```
!ready [Flags]
```

Parameters

Flags

Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default is 0x6.

Bit 1 (0x2)

Causes the display to include the thread's wait states.

Bit 2 (0x4)

If this is included without Bit 1 (0x2), this has no effect. If this is included along with Bit 1, the thread is displayed with a stack trace.

Bit 3 (0x8)

Causes the display of each function to include the return address and the stack pointer. The display of function arguments is suppressed.

Bit 4 (0x10)

Causes the display of each function to include only the return address; arguments and stack pointers are suppressed.

DLL

Kdexts.dll

Additional Information

For information about thread scheduling and the READY state, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The output from this extension is similar to that of [!thread](#), except that only ready threads are displayed, and they are sorted in order of decreasing priority.

!reg

Article • 10/25/2023

The **!reg** extension displays and searches through registry data.

```
dbgcmd

!reg {querykey|q} FullKeyPath
!reg keyinfo HiveAddress KeyNodeAddress
!reg kcb Address
!reg knode Address
!reg kbody Address
!reg kvalue Address
!reg valuelist HiveAddress KeyNodeAddress
!reg subkeylist HiveAddress KeyNodeAddress
!reg baseblock HiveAddress
!reg seccache HiveAddress
!reg hashindex [HiveAddress]HashKey
!reg openkeys {HiveAddress|0}
!reg openhandles {HiveAddress|0}
!reg findkcb FullKeyPath
!reg hivelist
!reg viewlist HiveAddress
!reg freebins HiveAddress
!reg freecells BinAddress
!reg dirtyvector HiveAddress
!reg cellindex HiveAddress Index
!reg freehints HiveAddress Storage Display
!reg translist {RmAddress|0}
!reg uowlist TransactionAddress
!reg locktable KcbAddress ThreadAddress
!reg convkey KeyPath
!reg postblocklist
!reg notifylist
!reg ixlock LockAddress
!reg dumppool [s|r]
```

Parameters

{querykey|q} **** *FullKeyPath*

Displays subkeys and values of a key if the key is cached. *FullKeyPath* specifies the full key path.

keyinfo *HiveAddress* **** *KeyNodeAddress*

Displays subkeys and values of a key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

kcb ** Address**

Displays a registry key control block. *Address* specifies the address of the key control block.

knode ** Address**

Displays a registry key node structure. *Address* specifies the address of the key node.

kbody ** Address**

Displays a registry key body structure. *Address* specifies the address of the key body. (Registry key bodies are the actual objects associated with handles.)

kvalue ** Address**

Displays a registry key value structure. *Address* specifies the address of the value.

valuelist ** HiveAddress **** KeyNodeAddress**

Displays a list of the values in the specified key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

subkeylist ** HiveAddress **** KeyNodeAddress**

Displays a list of the subkeys of the specified key node. *HiveAddress* specifies the address of the hive. *KeyNodeAddress* specifies the address of the key node.

baseblock ** HiveAddress**

Displays the base block for a hive (also known as the *hive header*). *HiveAddress* specifies the address of the hive.

seccache ** HiveAddress**

Displays the security cache for a hive. *HiveAddress* specifies the address of the hive.

hashindex ** [HiveAddress] **** HashKey**

Computes the hash index entry for a hash key. *HiveAddress* specifies the address of the hive. *HashKey* specifies the key.

Note *HiveAddress* is required if the target computer is running Windows 7 or later.

openkeys {HiveAddress|0}

Displays all open keys in a hive. *HiveAddress* specifies the address of the hive. If zero is used instead, the entire registry hash table is displayed; this table contains all open keys in the registry.

findkcb ** FullKeyPath**

Displays the registry key control block corresponding to a registry path. *FullKeyPath* specifies the full key path; this path must be present in the hash table.

hivelist

Displays a list of all hives in the system, along with detailed information about each hive.

viewlist ** *HiveAddress***

Displays all pinned and mapped views for a hive, with detailed information for each view. *HiveAddress* specifies the address of the hive.

freebins ** *HiveAddress***

Displays all free bins for a hive, with detailed information for each bin. *HiveAddress* specifies the address of the hive.

freecells ** *BinAddress***

Iterates through a bin and displays all free cells inside it. *BinAddress* specifies the address of the bin.

dirtyvector ** *HiveAddress***

Displays the dirty vector for a hive. *HiveAddress* specifies the address of the hive.

cellindex ** *HiveAddress* **** *Index***

Displays the virtual address for a cell in a hive. *HiveAddress* specifies the address of the hive. *Index* specifies the cell index.

freehints *HiveAddress* ** *Storage* **** *Display***

Displays free hint information.

translist {*RmAddress*|0}

Displays the list of active transactions in an RM. *RmAddress* specifies the address of the RM.

uowlist *TransactionAddress*

Displays the list of UoWs attached to a transaction. *TransactionAddress* specifies the address of the transaction.

locktable *KcbAddress ThreadAddress*

Displays relevant lock table content.

convkey *KeyPath*

Displays hash keys for a key path.

postblocklist

Displays the list of threads that have postblocks posted.

notifylist

Displays the list of notify blocks in the system.

!xlock LockAddress

Displays ownership of an intent lock. *LockAddress* specifies the address of the lock.

dumppool [s|r]

Displays registry-allocated paged pool. If **s** is specified, the list of registry pages is saved to a temporary file. If **r** is specified, the registry page list is restored from the previously saved temporary file.

DLL

Kdexts.dll

Additional Information

For information about the registry and its components, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example. First use **!reg hivelist** to get a list of hive addresses.

```
dbgcmd

00: kd> !reg hivelist
## 

## |      HiveAddr      |Stable Length|      Stable Map      |Volatile Length|
Volatile Map      |MappedViews|PinnedViews|U(Cnt)|      BaseBlock      |
FileName

| fffff8a000014010 |      1000 | fffff8a0000140b0 |      1000 |
fffff8a000014328 |      0| fffff8a00001e000 | <NONAME>
| fffff8a000028010 |     a15000 | fffff8a00002e000 |      1a000 |
fffff8a000028328 |      0| fffff8a000029000 | SYSTEM
| fffff8a00004f010 |     14000 | fffff8a00004f0b0 |      c000 |
fffff8a00004f328 |      0| fffff8a000050000 | <NONAME>
| fffff8a000329010 |      6000 | fffff8a0003290b0 |      0 |
0000000000000000 |      0| fffff8a00032f000 |

Device\HarddiskVolume1\Boot\BCD
| fffff8a0002f2010 |    4255000 | fffff8a0006fa000 |      6000 |
fffff8a0002f2328 |      0| fffff8a00036c000 |

emRoot\System32\Config\SOFTWARE
| fffff8a000df0010 |      f7000 | fffff8a000df00b0 |      1000 |
fffff8a000df0328 |      0| fffff8a000df1000 |

temRoot\System32\Config\DEFAULT
| fffff8a0010f8010 |      9000 | fffff8a0010f80b0 |      1000 |
fffff8a0010f8328 |      0| fffff8a0010f9000 |
```

```

emRoot\System32\Config\SECURITY
| fffff8a001158010 |      7000 | fffff8a0011580b0 |          0 |
0000000000000000 |      0| fffff8a001159000 |          0 |
\SystemRoot\System32\Config\SAM
| fffff8a00124b010 |     24000 | fffff8a00124b0b0 |          0 |
0000000000000000 |      0| fffff8a00124c000 |          0 |
files\NetworkService\NTUSER.DAT
| fffff8a0012df220 |     b7000 | fffff8a0012df2c0 |          0 |
0000000000000000 |      0| fffff8a0012e6000 |          0 |
\SystemRoot\System32\Config\BBI
| fffff8a001312220 |     26000 | fffff8a0013122c0 |          0 |
0000000000000000 |      0| fffff8a00117e000 |          0 |
rofiles\LocalService\NTUSER.DAT
| fffff8a001928010 |     64000 | fffff8a0019280b0 |      3000 |
fffff8a001928328 |      0| fffff8a00192b000 |          0 |
User.MYTESTCOMPUTER2\ntuser.dat
| fffff8a001b9b010 |    203000 | fffff8a001bc4000 |          0 |
0000000000000000 |      0| fffff8a001b9c000 |          0 |
\Microsoft\Windows\UsrClass.dat
| fffff8a001dc0010 |     30000 | fffff8a001dc00b0 |          0 |
0000000000000000 |      0| fffff8a001dc2000 | Volume
Information\Syscache.hve
## | fffff8a0022dc010 |    175000 | fffff8a0022dc0b0 |          0 |
0000000000000000 |      0| fffff8a0022dd000 |          0 |
\AppCompat\Programs\Amcache.hve

```

Use the third hive address in the preceding output (fffff8a00004f010) as an argument to !reg openkeys.

```

dbgcmd

0: kd> !reg openkeys fffff8a00004f010

# Hive: \REGISTRY\MACHINE\HARDWARE

Index e9: 3069276d kcb=fffff8a00007eb98 cell=00000220 f=00200000
\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM
Index 101: 292eea1f kcb=fffff8a00007ecc0 cell=000003b8 f=00200000
\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM\MULTIFUNCTIONADAPTER
Index 140: d927b0d4 kcb=fffff8a00007ea70 cell=000001a8 f=00200000
\REGISTRY\MACHINE\HARDWARE\DESCRIPTION
Index 160: 96d26a30 kcb=fffff8a00007e6f8 cell=00000020 f=002c0000
\REGISTRY\MACHINE\HARDWARE

# 0x4 keys found

```

Use the first full key path in the preceding output (\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM) as an argument to !reg querykey.

```

dbgcmd

```

```

0: kd> !reg querykey \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM

Found KCB = ffffff8a00007eb98 ::

\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM

Hive          ffffff8a00004f010
KeyNode       ffffff8a000054224

[SubKeyAddr]      [SubKeyName]
fffff8a000060244  CentralProcessor
fffff8a00006042c  FloatingPointProcessor
fffff8a0000543bc  MultifunctionAdapter

[SubKeyAddr]      [VolatileSubKeyName]
fffff8a000338d8c  BIOS
fffff8a0002a2e4c  VideoAdapterBusses

Use '!reg keyinfo ffffff8a00004f010 <SubKeyAddr>' to dump the subkey details

[ValueType]      [ValueName]          [ValueData]
REG_BINARY       Component Information 0x542AC - 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
REG_SZ           Identifier          AT/AT COMPATIBLE
REG_FULL_RESOURCE_DESCRIPTOR Configuration Data ff ff ff ff ff ff
ff ff 00 00 00 00 00 02 00 00 00 05 00 00 00 18 00 00 00 00 00 00 00 00 00
00 00 00 00 00 80 00 ff 03 00 00 3f 00 fe 00 02 00 81 00 fe 03 00 00 3f 00
fe 00 02 00 05 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 0c 00 00 00 04 00
REG_SZ           SystemBiosDate    07/18/07
REG_MULTI_SZ     SystemBiosVersion HPQOEM - 20070718\0\0
REG_SZ           VideoBiosDate    03/23/20
REG_MULTI_SZ     VideoBiosVersion Hardware Version 0.0\0\0

```

Here is another example:

```

dbgcmd

kd> !reg hivelist
##

## | HiveAddr | Stable Length|Stable Map|Volatile Length|Volatile
Map|MappedViews|PinnedViews|U(Cnt)| BaseBlock | FileName

| e16e7428 |      2000 | e16e7484 |          0 |  00000000 |      1
|      0 |      0| e101f000 | \Microsoft\Windows\UsrClass.dat
| e1705a78 |    77000 | e1705ad4 |      1000 | e1705bb0 |      30
|      0 |      0| e101c000 | ttings\Administrator\ntuser.dat
| e13d4b88 |   814000 | e146a000 |      1000 | e13d4cc0 |      255
|      0 |      0| e1460000 | emRoot\System32\Config\SOFTWARE
| e13ad008 |   23000 | e13ad064 |      1000 | e13ad140 |      9
|      0 |      0| e145e000 | temRoot\System32\Config\DEFAULT
| e13b3b88 |     a000 | e13b3be4 |      1000 | e13b3cc0 |      3

```

```
|      0 |      0| e145d000 | emRoot\System32\Config\SECURITY
| e142d008 |      5000 | e142d064 |          0 | 00000000 |          2
|      0 |      0| e145f000 | <UNKNOWN>
| e11e3628 |      4000 | e11e3684 |      3000 | e11e3760 |          0
|      0 |      0| e11e4000 | <NONAME>
| e10168a8 | 1c1000 | e1016904 |     15000 | e10169e0 |         66
|      0 |      0| e1017000 | SYSTEM
## | e10072c8 |      1000 | e1007324 |          0 | 00000000 |
0 |      0 |      0| e1010000 | <NONAME>
```

```
kd> !reg hashindex e16e7428
```

```
CmpCacheTable = e100a000
```

```
Hash Index[e16e7428] : 5ac
```

```
Hash Entry[e16e7428] : e100b6b0
```

```
kd> !reg openkeys e16e7428
```

```
Index 68: 7bab7683 kcb=e13314f8 cell=00000740 f=00200004 \REGISTRY\USER\S-
1-5-21-1715567821-413027322-527237240-500_Classes\CLSID
```

```
Index 7a1: 48a30288 kcb=e13a3738 cell=00000020 f=002c0004 \REGISTRY\USER\S-
1-5-21-1715567821-413027322-527237240-500_Classes
```

To display formatted registry key information, use the [!dreg](#) extension instead.

!regkcb

Article • 10/25/2023

The **!regkcb** extension displays a registry key control block.

dbgcmd

!regkcb Address

Parameters

Address

Specifies the address of the key control block.

DLL

Windows 2000	Kdextx86.dll
Windows XP and later	Unavailable

Additional Information

For information about the registry and its components, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

In Windows 2000, **!regkcb** displays a specific registry key control block.

In Windows XP and later versions of Windows, the [!reg](#) extension command should be used instead.

Every registry key has a control block that contains properties, such as its permissions.

!rellist

Article • 10/25/2023

The **!rellist** extension displays a Plug and Play relation list.

dbgcmd

```
!rellist Address [Flags]
```

Parameters

Address

Specifies the address of the relation list.

Flags

Specifies additional information to include in the display. This can be any combination of the following bits (the default is zero):

Bit 1 (0x2)

Causes the display to include the CM_RESOURCE_LIST. The display also includes the boot resources list, if it is available.

Bit 2 (0x4)

Causes the display to include the resource requirements list (IO_RESOURCE_LIST).

Bit 3 (0x8)

Causes the display to include the translated CM_RESOURCE_LIST.

DLL

Kdexts.dll

Additional Information

See [Plug and Play Debugging](#) for applications of this extension command. For information about these list structures, see the Windows Driver Kit (WDK) documentation.

!ruleinfo

Article • 10/25/2023

The **!ruleinfo** command displays information about a Driver Verifier rule.

```
dbgcmd
```

```
!ruleinfo RuleId [RuleState [SubState]]
```

Parameters

RuleId

The ID of the verifier rule. This is the first argument of the [DRIVER_VERIFIER_DETECTED_VIOLATION](#) bug check.

RuleState

Additional state information about the violation. This is the third argument of the [DRIVER_VERIFIER_DETECTED_VIOLATION](#) bug check.

SubState

Sub-state information about the violation. This is the fourth argument of the [DRIVER_VERIFIER_DETECTED_VIOLATION](#) bug check.

DLL

ext.dll

Remarks

This command applies only to rules in the Driver Verifier extension; that is, rules that have an ID greater than or equal to 0x10000.

The following example shows the four arguments of a [DRIVER_VERIFIER_DETECTED_VIOLATION](#) bug check.

```
dbgcmd
```

```
DRIVER_VERIFIER_DETECTED_VIOLATION (c4)
```

```
...
```

```
Arguments:
```

```
Arg1: 0000000000091001, ID of the 'NdisOidComplete' rule that was violated.
```

```
Arg2: fffff800002d49d0, A pointer to the string describing the violated rule
```

```
condition.  
Arg3: fffffe000027b8370, Address of internal rule state (second argument to  
!ruleinfo).  
Arg4: fffffe000027b83f8, Address of supplemental states (third argument to  
!ruleinfo).  
  
## Debugging Details:  
  
DV_VIOLATED_CONDITION: This OID should only be completed with  
NDIS_STATUS_NOT_ACCEPTED,  
NDIS_STATUS_SUCCESS, or NDIS_STATUS_PENDING.  
  
DV_MSDN_LINK: https://go.microsoft.com/fwlink/?linkid=278802  
  
DRIVER_OBJECT: fffffe0000277a2b0  
...  
  
STACK_TEXT:  
fffffd000`2118ff58 ffffff803`4c83afa2 : 00000000`000000c4 00000000`00000001  
...  
fffffd000`2118ff60 ffffff803`4c83a8c0 : 00000000`00000003 00000000`00091001  
...  
...  
  
STACK_COMMAND: kb  
  
FOLLOWUP_NAME: XXXX  
  
FAILURE_BUCKET_ID: XXXX  
...
```

In the preceding output, the rule ID (0x91001) is shown as Arg1. Arg3 and Arg4 are the addresses of rule state and substate information. You can pass the rule ID, the rule state, and the substate to !ruleinfo to get a description of the rule and a link to detailed documentation of the rule.

```
dbgcmd  
  
3: kd> !ruleinfo 0x91001 0xfffffe000027b8370 0xfffffe000027b83f8  
  
RULE_ID: 0x91001  
  
RULE_NAME: NdisOidComplete  
  
RULE_DESCRIPTION:  
This rule verifies if an NDIS miniport driver completes an OID correctly.  
Check RULE_STATE for Oid ( use !ndiskd.oid ), which can be one of the  
following:  
1) NULL,  
2) Pending OID, or  
3) Previous OID if no OID is pending.
```

MSDN_LINK: <https://learn.microsoft.com/windows-hardware/drivers/devtest/ndis-ndisoidcomplete>

CONTEXT: Miniport 0xFFFFE0000283F1A0

CURRENT_TIME (Timed Rules): 142 seconds

RULE_STATE: 0xFFFFE000027B83F8

!running

Article • 04/03/2024

The **!running** extension displays a list of running threads on all processors of the target computer.

```
dbgcmd
```

```
!running [-i] [-t]
```

Parameters

-i

Causes the display to include idle processors as well.

-t

Causes a stack trace to be displayed for each processor.

DLL

Kdexts.dll

Additional Information

For more information about debugging multiprocessor computers, see [Multiprocessor Syntax](#).

Remarks

With no options, **!running** will display the affinity of all active processors and all idle processors. For all active processors, it will also display the current and next thread fields from the process control block (PRCB) and the state of the 16 built-in queued spin locks.

Here is an example from a multiprocessor Itanium system:

```
dbgcmd
```

```
0: kd> !running
```

```
System Processors 3 (affinity mask)
```

Idle Processors 0

	Prcb	Current	Next	
0	e0000000818f8000	e0000000818f9e50	e0000000866f12f0
1	e000000086f16010	e00000008620eb0	e000000086eddbc0	.0.....

The 16 characters at the end of each line indicate the built-in queued spin locks (the LockQueue entries in the PRCB). A period (.) indicates that the lock is not in use, **O** means the lock is owned by this processor, and **W** means the processor is queued for the lock. To see more information about the spin lock queue, use [!qlocks](#).

Here is an example that shows active and idle processors, along with their stack traces:

```
dbgcmd

0: kd> !running -it

System Processors f (affinity mask)
  Idle Processors f
All processors idle.

      Prcb      Current    Next
      0  ffdff120  805495a0          .....
      ChildEBP RetAddr
      8053e3f0 805329c2 nt!RtlpBreakWithStatusInstruction
      8053e3f0 80533464 nt!_KeUpdateSystemTime+0x126
      ffdff980 ffdff980 nt!KiIdleLoop+0x14

      1  f87e0120  f87e2e60          .....
      ChildEBP RetAddr
      f87e0980 f87e0980 nt!KiIdleLoop+0x14

      2  f87f0120  f87f2e60          .....
      ChildEBP RetAddr
      f87f0980 f87f0980 nt!KiIdleLoop+0x14

      3  f8800120  f8802e60          .....
      ChildEBP RetAddr
      f8800980 f8800980 nt!KiIdleLoop+0x14
```

Feedback

Was this page helpful?

 Yes

 No

!scm

Article • 04/03/2024

The **!scm** extension displays the specified shared cache map.

In Windows XP and later versions of Windows, use the [dt nt!_SHARED_CACHE_MAP Address](#) command instead of **!scm**.

```
dbgcmd
!scm Address
```

Parameters

Address

Specifies the address of the shared cache map.

DLL

Unavailable

Additional Information

For information about cache management, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

For information about other cache management extensions, see the [!cchelp](#) extension.

Feedback

Was this page helpful?

 Yes

 No

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

!search

Article • 10/25/2023

The **!search** extension searches pages in physical memory for pointer-sized data that matches the specified criteria.

Syntax

dbgcmd

```
!search [-s] [-p] Data [ Delta [ StartPFN [ EndPFN ]]]  
!search -?
```

Parameters

-s

Causes symbol check errors to be ignored during the search. This is useful if you are getting too many "incorrect symbols for kernel" errors.

-p

Causes the value of *Data* to be interpreted as a 32-bit value, preventing any sign extension.

Data

Specifies the data to search for. *Data* must be the size of a pointer on the target system (32 bits or 64 bits). An exact match for the value of *Data* is always displayed. Other matches are displayed as well, depending on the value of *Delta*; see the Remarks section below for details.

Delta

Specifies the allowable difference between a value in memory and the value of *Data*. See the Remarks section below for details.

StartPFN

Specifies the page frame number (PFN) of the beginning of the range to be searched. If this is omitted, the search begins at the lowest physical page.

EndPFN

Specifies the page frame number (PFN) of the end of the range to be searched. If this is omitted, the search ends at the highest physical page.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For more ways to display and search physical memory, see [Reading and Writing Memory](#).

Remarks

If *StartPFN* and *EndPFN* are specified, these are taken as the page frame numbers of the beginning and end of the range in physical memory to be searched. For an explanation of page frame numbers, see [Converting Virtual Addresses to Physical Addresses](#). If *StartPFN* and *EndPFN* are omitted, all physical memory is searched.

All hits are displayed.

The **!search** extension will search through all memory for in the specified page range and examine each ULONG_PTR-aligned value. Values that satisfy at least one of the following criteria are displayed:

- The value matches *Data* exactly.
- If Delta is 0 or omitted: The value differs from *Data* by a single bit.
- If Delta is nonzero: The value differs from *Data* by at most *Delta*. In other words, the value lies in the range [Data - Delta, Data + Delta].
- If Delta is nonzero: The value differs from the lowest number in the range (Data - Delta) by a single bit.

In most cases, *Data* will specify an address you are interested in, but any ULONG_PTR sized data can be specified.

Because the debugger's search engine structures reside in memory on the target computer, if you search all of memory (or any range containing these structures) you will see matches in the area where the structures themselves are located. If you need to eliminate these matches, do a search for a random value; this will indicate where the debugger's search structures are located.

Here are some examples. The following will search the memory page with PFN 0x237D for values between 0x80001230 and 0x80001238, inclusive:

```
dbgcmd  
kd> !search 80001234 4 237d 237d
```

The following will search the memory pages ranging from PFN 0x2370 to 0x237F for values that are within one bit of 0xF100F0F. The exact matches are indicated in bold; the others are off by one bit:

```
dbgcmd  
kd> !search 0f100f0f 0 2370 237f  
Searching PFNs in range 00002370 - 0000237F for [0F100F0F - 0F100F0F]  
  
Pfn      Offset     Hit      Va      Pte  
- - - - -  
0000237B 00000368 0F000F0F 01003368 C0004014  
0000237C 00000100 0F100F0F 01004100 C0004014  
0000237D 000003A8 0F100F0F 010053A8 C0004014  
0000237D 000003C8 0F100F8F 010053C8 C0004014  
0000237D 000003E8 0F100F0F 010053E8 C0004014  
0000237D 00000408 0F100F0F 01005408 C0004014  
0000237D 00000428 0F100F8F 01005428 C0004014  
Search done.
```

The columns in the display are as follows: **Pfn** is the page frame number (PFN) of the page; **Offset** is the offset on that page; **Hit** is the value at that address; **Va** is the virtual address mapped to this physical address (if this exists and can be determined); **Pte** is the page table entry (PTE).

To calculate the physical address, shift the PFN left three hexadecimal digits (12 bits) and add the offset. For example, the last line in the table is virtual address 0x0237D000 + 0x428 = 0x0237D428.

!searchpte

Article • 04/03/2024

The **!searchpte** extension searches physical memory for the specified page frame number (PFN).

dbgcmd

```
!searchpte PFN  
!searchpte -?
```

Parameters

PFN

Specifies the PFN in hexadecimal format.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To stop execution at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!sel

Article • 04/01/2024

The `!sel` extension command is obsolete. Use the [dg \(Display Selector\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!session

Article • 10/25/2023

The **!session** extension controls the session context. It can also display a list of all user sessions.

Syntax

```
dbgcmd  
!session  
!session -s DefaultSession  
!session -?
```

Parameters

-s ** *DefaultSession***

Sets the [session context](#) to the specified value. If *DefaultSession* is -1, the session context is set to the current session.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For information about user sessions and the Session Manager (smss.exe), see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!session** extension is used to control the session context. Using **!session** with no parameters will display a list of active sessions on the target computer. Using **!session /s *DefaultSession*** will change the session context to the new default value.

When you set the session context, the process context is automatically changed to the active process for that session, and the [.cache forcedecodeptes](#) option is enabled so

that session addresses are translated properly.

For more details and a list of all the session-related extensions that are affected by the session context, see [Changing Contexts](#).

!smt

Article • 04/03/2024

The **!smt** extension displays a summary of the simultaneous multithreaded processor information.

```
dbgcmd
```

```
!smt
```

DLL

Kdexts.dll

Remarks

Here is an example:

```
dbgcmd
```

```
1kd> !smt
SMT Summary:
-----
KeActiveProcessors: **----- (00000003)
KiIdleSummary: ----- (00000000)
No PRCB      Set Master SMT Set          IAID
 0 820f4820  Master    **----- (00000003)  00
 1 87a4d120  820f4820  **----- (00000003)  01

Maximum cores per physical processor:  2
Maximum logical processors per core:   1
```

The **No** column indicates the number of the processor.

The **PRCB** column indicates the address of the processor control block for the processor. Each logical processor is listed separately.

Each physical processor has exactly one logical processor listed as the **Master** under the **Set Master** column.

The **SMT Set** column lists the processor's simultaneous multithreaded processor set information.

The IAID column lists the initial Advanced Programmable Interrupt Controller identifier (APIC ID). On a true x64 computer, each processor starts with a hard-coded initial APIC ID. This ID value can be retrieved through the CPUID instruction. On certain other computers, the initial APIC ID is not necessarily unique across all processors, so the APIC ID that is accessible through the APIC's memory-mapped input/output (MMIO) space can be modified. This technique enables software to allocate unique APIC IDs for all processors within the computer. Depending on the target computer's processors, the IAID column may show this ID or may be blank.

Feedback

Was this page helpful?

 Yes

 No

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

!sprocess

Article • 04/03/2024

The **!sprocess** extension displays information about the specified session process, or about all processes in the specified session.

dbgcmd

```
!sprocess Session [Flags [ImageName]]  
!sprocess -?
```

Parameters

Session

Specifies the session that owns the desired process. *Session* is always interpreted as a decimal number.

Session can have the following values:

[+] Expand table

-1	Use current session. This is the default.
-2	Use session context .
-4	Display all processes by session.

Flags

Specifies the level of detail in the display. *Flags* can be any combination of the following bits. The default is 0.

[+] Expand table

0x0	Display minimal information.
Bit 0 (0x1)	Display time and priority statistics.
Bit 1 (0x2)	Adds to the display a list of threads and events associated with the process and the wait states of the threads.
Bit 2 (0x4)	Adds to the display a list of threads associated with the process. If this bit is used without Bit 1 (0x2), each thread is displayed on a single line.

	If this is included with Bit 1, each thread is displayed with a stack trace.
Bit 3 (0x8)	Adds to the display of each function the return address, the stack pointer and, on Itanium-based systems, the bsp register value. It suppresses the display of function arguments.
Bit 4 (0x10)	Display only the return address of each function. Suppress the arguments and stack pointers.

ImageName

Specifies the name of the process to be displayed. All processes whose executable image names match *ImageName* are displayed. The image name must match that in the EPROCESS block. In general, this is the executable name that was invoked to start the process, including the file extension (usually .exe), and truncated after the fifteenth character. There is no way to specify an image name that contains a space.

-?

Displays help for this extension in the Debugger Command window. This help text has some omissions.

DLL

Kdexts.dll

Additional Information

For information about sessions and processes in kernel mode, see [Changing Contexts](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The output of this extension is similar to that of [!process](#), except that the addresses of _MM_SESSION_SPACE and _MMSESSION are displayed as well.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!srb

Article • 10/25/2023

The **!srb** extension displays information about a SCSI Request Block (SRB).

dbgcmd

!srb Address

Parameters

Address

Specifies the hexadecimal address of the SRB on the target computer.

DLL

Kdexts.dll

Additional Information

For information about SRBs, see the Windows Driver Kit (WDK) documentation.

Remarks

An SRB is a system-defined structure used to communicate I/O requests from a SCSI class driver to a SCSI port driver.

!stacks

Article • 10/25/2023

The **!stacks** extension displays information about the kernel stacks.

Syntax

```
dbgcmd  
!stacks [Detail [FilterString]]
```

Parameters

Detail

Specifies the level of detail to use in the display. The following table lists the valid values for *Detail*.

0	Displays a summary of the current kernel stacks. This is the default value.
1	Displays stacks that are currently paged out, as well as the current kernel stacks.
2	Displays the full parameters for all stacks, as well as stacks that are currently paged out and the current kernel stacks.

FilterString

Displays only threads that contain the specified substring in a symbol.

DLL

Kdexts.dll

Additional Information

For information about kernel stacks, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The **!stacks** extension gives a brief summary of the state of every thread. You can use this extension instead of the **!process** extension to get a quick overview of the system, especially when debugging multithread issues such as resource conflicts or deadlocks.

The **!findstack** user-mode extension also displays information about particular stacks.

Here is an example of the simplest **!stacks** display:

```
dbgcmd

kd> !stacks 0
Proc.Thread .Thread ThreadState Blocker
              [System]
4.000050 827eea10 Blocked    +0xfe0343a5
                                         [smss.exe]

                                         [csrss.exe]
b0.0000a8 82723b70 Blocked    ntoskrnl!_KiSystemService+0xc4
b0.0000c8 82719620 Blocked    ntoskrnl!_KiSystemService+0xc4
b0.0000d0 827d5d50 Blocked    ntoskrnl!_KiSystemService+0xc4
....
```

The first column shows the process ID and the thread ID (separated by a period).

The second column is the current address of the thread's ETHREAD block.

The third column shows the state of the thread (initialized, ready, running, standby, terminated, transition, or blocked).

The fourth column shows the top address on the thread's stack.

Here are examples of more detailed **!stacks** output:

```
dbgcmd

kd> !stacks 1
Proc.Thread .Thread ThreadState Blocker
              [System]
4.000008 827d0030 Blocked    ntoskrnl!MmZeroPageThread+0x66
4.000010 827d0430 Blocked    ntoskrnl!ExpWorkerThread+0x189
4.000014 827cf030 Blocked    Stack paged out
4.000018 827cfda0 Blocked    Stack paged out
4.00001c 827cfb10 Blocked    ntoskrnl!ExpWorkerThread+0x189
.....
                                         [smss.exe]
9c.000098 82738310 Blocked    Stack paged out
9c.0000a0 826a5190 Blocked    Stack paged out
9c.0000a4 82739d30 Blocked    Stack paged out
```

```
[csrss.exe]
b0.0000bc 826d0030 Blocked Stack paged out
b0.0000b4 826c9030 Blocked Stack paged out
b0.0000a8 82723b70 Blocked ntoskrnl!_KiSystemService+0xc4
.....
kd> !stacks 2
Proc.Thread .Thread ThreadState Blocker
[System]
4.000008 827d0030 Blocked ntoskrnl!KiSwapThread+0xc5
ntoskrnl!KeWaitForMultipleObjects+0x2b4
ntoskrnl!MmZeroPageThread+0x66
ntoskrnl!Phase1Initialization+0xd82
ntoskrnl!PspSystemThreadStartup+0x4d
ntoskrnl!CreateSystemRootLink+0x3d8
+0x3f3f3f3f
4.000010 827d0430 Blocked ntoskrnl!KiSwapThread+0xc5
ntoskrnl!KeRemoveQueue+0x191
.....
```

!swd

Article • 04/03/2024

The **!swd** extension displays the software watchdog timer states for the specified processor, including the deferred procedure call (DPC) and the watchdog timer states for threads.

```
dbgcmd
!swd [Processor]
```

Parameters

Processor

Specifies the processor. If *Processor* is omitted, information is displayed for all processors on the target computer.

DLL

Kdexts.dll

Remarks

The watchdog timer shuts down or restarts Windows if Windows stops responding. The times are displayed in seconds.

Feedback

Was this page helpful?

 Yes

 No

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

!sysinfo

Article • 04/03/2024

The **!sysinfo** extension reads and displays specified SMBIOS, Advanced Configuration and Power Interface (ACPI), and CPU information from a dump file or live system.

dbgcmd

```
!sysinfo cpuinfo [-csv [-noheaders]]  
!sysinfo cpumicrocode [-csv [-noheaders]]  
!sysinfo cpuspeed [-csv [-noheaders]]  
!sysinfo gbl [-csv [-noheaders]]  
!sysinfo machineid [-csv [-noheaders]]  
!sysinfo registers  
!sysinfo smbios [-csv [-noheaders]] {-debug | -devices | -memory | -power |  
-processor | -system | -v}  
!sysinfo -?
```

Parameters

cpuinfo

Displays information about the processor.

cpumicrocode

(GenuineIntel processors only) Displays the initial and cached microcode processor versions.

cpuspeed

Displays the maximum and current processor speeds.

gbl

Displays the BIOS list of ACPI tables.

machineid

Displays machine ID information for the SMBIOS, BIOS, firmware, system, and baseboard.

registers

Displays machine-specific registers (MSRs).

smbios

Displays the SMBIOS table.

-CSV

Displays all data in comma-separated, variable-length (CSV) format.

-noheaders

Suppresses the header for the CSV format.

-debug

Displays output in standard format and CSV format.

-devices

Displays the device entries in the SMBIOS table.

-memory

Displays the memory entries in the SMBIOS table.

-power

Displays the power entries in the SMBIOS table.

-processor

Displays the processor entries in the SMBIOS table.

-system

Displays the system entries in the SMBIOS table.

-v

Verbose. Displays the details of entries in the SMBIOS table.

-?

Displays help for this extension in the Debugger Command window.

DLL

Kdexts.dll

Remarks

This extension is useful only when the dump file is a System Crash File (.dmp) that has not been converted to a minidump file from a kernel or full dump file, or the live system has finished starting and is online (for example, at the log-in prompt).

You can use any combination of the **-debug**, **-devices**, **-memory**, **-power**, **-processor**, **-system**, and **-v** parameters in a single extension command.

The following parameters are supported only on particular systems:

- The **gbl** parameter works only when the target computer supports ACPI.
- The **smbios** parameter works only when the target computer supports SMBIOS.

Microsoft makes every effort to remove personally identifiable information (PII) from these records. All PII is removed from dump files. However, on a live system, some PII may not yet be removed. As a result, PII fields will be reported as 0 or blank, even if they actually contain information.

To stop execution of commands that include the **cpuinfo**, **gbl**, **registers**, or **smbios** parameters at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

Feedback

Was this page helpful?



Yes



No

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

!sysptes

Article • 10/25/2023

The **!sysptes** extension displays a formatted view of the system page table entries (PTEs).

```
dbgcmd
```

```
!sysptes [Flags]
```

Parameters

Flags

Specifies the level of detail to display. *Flags* can be any combination of the following bits. The default is zero:

Bit 0 (0x1)

Displays information about free PTEs.

Bit 1 (0x2)

(Windows 2000 only) Displays unused pages in the page usage statistics.

Displays information about free PTEs in the global special pool.

Bit 2 (0x4)

Displays detailed information about any system PTEs that are allocated to mapping locked pages.

Bit 3 (0x8)

(Windows 2000 and Windows XP only) Displays nonpaged pool expansion free PTE information. If this bit is set, the other lists are not displayed. If both 0x1 and 0x8 are set, all nonpaged pool expansion free PTEs are displayed. If only 0x8 is set, only the total is displayed.

Bit 4 (0x10)

(Windows Vista and later) Displays special pool free PTE information for the session.

DLL

Kdexts.dll

Additional Information

For information about page tables and PTEs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To examine a specific PTE, use the [!pte](#) extension.

Here is an example from a Windows system:

```
dbgcmd

kd> !sysptes 1

System PTE Information
Total System Ptes 571224
  SysPtes list of size 1 has 361 free
  SysPtes list of size 2 has 91 free
  SysPtes list of size 4 has 48 free
  SysPtes list of size 8 has 36 free
  SysPtes list of size 9 has 29 free
  SysPtes list of size 23 has 29 free

starting PTE: ffffffe0059388000
ending PTE:    ffffffe00597e3ab8

  free ptes: ffffffe0059388000  number free: 551557.
  free ptes: ffffffe00597be558  number free: 104.
  free ptes: ffffffe00597d2828  number free: 676.

free blocks: 3  total free: 552337    largest free block: 551557
```

!thread

Article • 10/25/2023

The **!thread** extension displays summary information about a thread on the target system, including the ETHREAD block. This command can be used only during kernel-mode debugging.

This extension command is not the same as the [.thread \(Set Register Context\)](#) command.

Syntax

dbgcmd

```
!thread [-p] [-t] [Address [Flags]]
```

Parameters

-p

Displays summary information about the process that owns the thread.

-t

When this option is included, *Address* is the thread ID, not the thread address.

Address

Specifies the hexadecimal address of the thread on the target computer. If *Address* is -1 or omitted, it indicates the current thread.

Flags

Specifies the level of detail to display. *Flags* can be any combination of the following bits. If *Flags* is 0, only a minimal amount of information is displayed. The default is 0x6:

Bit 1 (0x2)

Displays the thread's wait states.

Bit 2 (0x4)

If this bit is used without Bit 1 (0x2), it has no effect. If this bit is used with Bit 1, the thread is displayed with a stack trace.

Bit 3 (0x8)

Adds the return address, the stack pointer, and (on Itanium systems) the **bsp** register

value to the information displayed for each function and suppresses the display of function arguments.

Bit 4 (0x10)

Sets the process context equal to the process that owns the specified thread for the duration of this command. This results in more accurate display of thread stacks.

DLL

Kdexts.dll

Additional Information

For information about threads in kernel mode, see [Changing Contexts](#) and [Controlling Processes and Threads](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich, Alex Ionescu and David Solomon.

Remarks

Here is an example using Windows 10:

```
dbgcmd

0: kd> !thread 0xffffcb088f0a4480
THREAD fffffcb088f0a4480 Cid 0e34.3814 Teb: 0000001a27ca6000 Win32Thread:
0000000000000000 RUNNING on processor 0
Not impersonating
DeviceMap          fffffb80842016c20
Owning Process    fffffcb08905397c0      Image:      MsMpEng.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 182835891      Ticks: 0
Context Switch Count 5989        IdealProcessor: 3
UserTime           00:00:01.046
KernelTime         00:00:00.296
Win32 Start Address 0x00007ffb3b2fd1b0
Stack Init fffff95818476add0 Current fffff958184769d30
Base fffff95818476b000 Limit fffff958184765000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr          : Args to Child
: Call Site
fffff802`59858c68 fffff801`b56d24aa : fffffcb08`8fd68010 00000000`00000000
fffff802`58259600 00000000`00000008 : nt!DbgBreakPointWithStatus
[d:\rs2\minkernel\ntos\rtl\amd64\debugstb.asm @ 130]
fffff802`59858c70 fffffcb08`8fd68010 : 00000000`00000000 fffff802`58259600
00000000`00000008 fffffcb08`8f0a4400 : 0xfffffff801`b56d24aa
```

```
fffff802`59858c78 00000000`00000000 : fffff802`58259600 00000000`00000008  
fffffcb08`8f0a4400 00000000`00000019 : 0xfffffcb08`8fd68010
```

Use commands like **!process** to locate the address or thread ID of the thread you are interested in.

The useful information in the **!thread** display is explained in the following table.

Parameter	Meaning
Thread address	The hexadecimal number after the word <i>THREAD</i> is the address of the ETHREAD block. In the preceding example, the thread address is 0xfffffcb088f0a4480 .
Thread ID	The two hexadecimal numbers after the word <i>Cid</i> are the process ID and the thread ID: <i>process ID.thread ID</i> . In the preceding example, the process ID is 0x0e34, and the thread ID is 0x3814.
Thread Environment Block (TEB)	The hexadecimal number after the word <i>Teb</i> is the address of the thread environment block (TEB).
Win32Thread	The hexadecimal number after the word <i>Win32Thread</i> is the address of the Win32Thread.
Thread State	The thread state is displayed at the end of the line that begins with the word <i>RUNNING</i> .
Owning Process	The hexadecimal number after the words <i>Owning Process</i> is the address of the EPROCESS for the process that owns this thread.
Start Address	The hexadecimal number after the words <i>Start Address</i> is the thread start address. This might appear in symbolic form.
User Thread Function	The hexadecimal number after the words <i>Win32 Start Address</i> is the address of the user thread function.
Priority	The priority information for the thread follows the word <i>Priority</i> .
Stack trace	A stack trace for the thread appears at the end of this display.

!threadfields

Article • 04/03/2024

The **!threadfields** extension command is not available. Instead, use **dt** (Display Type)..

```
dbgcmd
```

```
!threadfields
```

DLL

Unavailable (see the Remarks section)

Additional Information

For information about the ETHREAD block, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension command is not available. Instead, use the [dt \(Display Type\)](#) command to show the ETHREAD structure directly:

```
dbgcmd
```

```
kd> dt nt!_ETHREAD
```

Feedback

Was this page helpful?

 Yes

 No

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

!time

Article • 10/25/2023

The **!time** extension command displays information about the system time and interrupt time and can operate on explicit time stamps and the current time.

Environment

Item	Description
Modes	User mode, kernel mode
Targets	Live, crash dump
Platforms	All

Remarks

Here is an example in user mode:

```
dbgcmd

0:000> !time
CURRENT TIME:
System:          01d75f1c`8588b6ee (2021 Jun 11 23:50:13.477)
Interrupt:        0000020e`22112c46 (2 days, 14:46:12.434)
```

See also

[.time \(Display System Time\)](#)

[.echotime \(Show Current Time\)](#)

[.echotimestamps \(Show Time Stamps\)](#)

[!runaway](#)

!timer

Article • 10/25/2023

The **!timer** extension displays a detailed listing of all system timer use.

dbgcmd

!timer

PLL

Kdexts.dll

Additional Information

For information about timer objects, see the Windows Driver Kit (WDK) documentation.

Remarks

The `!timer` extension displays the timer tree, which stores all timer objects in the system.

Here is an example:

dbgcmd

```
kd> !timer
```

Dump system timers

Interrupt time: 9f760774 00000000 [12/ 8/2000 10:59:22.685 (Pacific Standard Time)]

List	Timer	Interrupt	Low/High	Fire Time	DPC/thread
0	8016aea0	P	9fb d8e00	00000000 [12/ 8/2000 10:59:23.154]	
	ntoskrnl!PopScanIdleList				
1	8257f118	e4e4225a	00000000 [12/ 8/2000 11:01:19.170]	thread 8257f030	
3	80165fc0	286be1c9	0000594a [4/ 1/2001 01:59:59.215]		
	ntoskrnl!ExpTimeZoneDpcRoutine				
	80165f40	2a7bf8d9	006f105e [12/31/2099 23:59:59.216]		
	ntoskrnl!ExpCenturyDpcRoutine				
5	825a0bf8	a952e1c2	00000000 [12/ 8/2000 10:59:39.232]	thread 825a0b10	
10	8251c7a8	41f54d84	00000001 [12/ 8/2000 11:03:55.310]	thread 8251c6c0	
	8249fe88	41f54d84	00000001 [12/ 8/2000 11:03:55.310]	thread 8249fda0	
11	8250e7e8	bc73ffde	00000000 [12/ 8/2000 11:00:11.326]	thread 8250e700	

• • • • •

```
237 82757070 9f904152 00000000 [12/ 8/2000 10:59:22.857] +f7a56f2e
    82676348 9f904152 00000000 [12/ 8/2000 10:59:22.857] +fe516352
    82728b78 9f904152 00000000 [12/ 8/2000 10:59:22.857] +fe516352
238 fe4b5d78 9f92a3ac 00000000 [12/ 8/2000 10:59:22.873] thread 827ceb10
    801658f0 9f92a3ac 00000000 [12/ 8/2000 10:59:22.873]

ntoskrnl!CcScanDpc
```

```
239 8259ad40 765a6f19 00000bba [12/23/2000 09:07:22.900] thread 825d3670
250 826d42f0 1486bed8 80000000 [           NEVER      ] thread 825fa030
```

Total Timers: 193, Maximum List: 7

Current Hand: 226, Maximum Search: 0

Wakeable timers:

!tokenfields

Article • 04/03/2024

The **!tokenfields** extension is obsolete. Instead, use the [dt \(Display Type\)](#) command.

```
dbgcmd
```

```
!tokenfields
```

DLL

Unavailable (see the Remarks section)

Additional Information

For information about the TOKEN structure, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon. This book may not be available in some languages and countries.(The user-mode token structures described in the Microsoft Windows SDK documentation are slightly different.)

Remarks

This extension command is not available in Windows XP or later versions of Windows. Instead, use the [dt \(Display Type\)](#) command to show the TOKEN structure directly:

```
dbgcmd
```

```
kd> dt nt!_TOKEN
```

To see a specific instance of the TOKEN structure, use the [!token](#) extension.

Here is an example of **!tokenfields** from a Windows 2000 system:

```
dbgcmd
```

```
kd> !tokenfields
TOKEN structure offsets:
TokenSource:          0x0
AuthenticationId:    0x18
ExpirationTime:      0x28
ModifiedId:          0x30
UserAndGroupCount:   0x3c
```

PrivilegeCount:	0x44
VariableLength:	0x48
DynamicCharged:	0x4c
DynamicAvailable:	0x50
DefaultOwnerIndex:	0x54
DefaultDacl:	0x6c
TokenType:	0x70
ImpersonationLevel:	0x74
TokenFlags:	0x78
TokenInUse:	0x79
ProxyData:	0x7c
AuditData:	0x80
VariablePart:	0x84

Feedback

Was this page helpful?

 Yes

 No

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

!trap

Article • 04/01/2024

The `!trap` extension command is obsolete. Use the [.trap \(Display Trap Frame\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!tss

Article • 04/01/2024

The **!tss** extension command is obsolete. Use the [.tss \(Display Task State Segment\)](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!tz

Article • 10/25/2023

The **!tz** extension displays the specified power thermal zone structure.

dbgcmd

!tz [Address]

Parameters

Address

The address of a power thermal zone that you want to display. If this parameter is omitted, the display includes all thermal zones on the target computer.

DLL

Kdexts.dll

Additional Information

To view the system's power capabilities, use the [!pocaps](#) extension command. To view the system's power policy, use the [!popolicy](#) extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To stop execution at any time, press CTRL+BREAK (in WinDbg) or CTRL+C (in KD).

!tzinfo

Article • 10/25/2023

The **!tzinfo** extension displays the contents of the specified thermal zone information structure.

```
dbgcmd
```

```
!tzinfo Address
```

Parameters

Address

The address of a thermal zone information structure that you want to display.

DLL

Kdexts.dll

Additional Information

To view the system's power capabilities, use the [!pocaps](#) extension command. To view the system's power policy, use the [!popolicy](#) extension command. For information about power capabilities and power policy, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

!ubc

Article • 10/25/2023

The **!ubc** extension clears a user-space breakpoint.

```
dbgcmd
```

```
!ubc BreakpointNumber
```

Parameters

BreakpointNumber

Specifies the number of the breakpoint to be cleared. An asterisk (*) indicates all breakpoints.

DLL

Kdexts.dll

Remarks

This will permanently delete a breakpoint set with [!ubp](#).

See also

[!ubd](#)

[!ube](#)

[!ubl](#)

[!ubp](#)

[User Space and System Space](#)

!ubd

Article • 10/25/2023

The **!ubd** extension temporarily disables a user-space breakpoint.

```
dbgcmd
```

```
!ubd BreakpointNumber
```

Parameters

BreakpointNumber

Specifies the number of the breakpoint to be disabled. An asterisk (*) indicates all breakpoints.

DLL

Kdexts.dll

Remarks

Disabled breakpoints will be ignored. Use [!ube](#) to re-enable the breakpoint.

See also

[!ubc](#)

[!ube](#)

[!ubl](#)

[!ubp](#)

[User Space and System Space](#)

!ube

Article • 10/25/2023

The **!ube** extension re-enables a user-space breakpoint.

```
dbgcmd
```

```
!ube BreakpointNumber
```

Parameters

BreakpointNumber

Specifies the number of the breakpoint to be enabled. An asterisk (*) indicates all breakpoints.

DLL

Kdexts.dll

Remarks

This is used to re-enable a breakpoint that was disabled by [!ubd](#).

See also

[!ubc](#)

[!ubd](#)

[!ubl](#)

[!ubp](#)

[User Space and System Space](#)

!ubl

Article • 10/25/2023

The **!ubl** extension lists all user-space breakpoints and their current status.

```
dbgcmd
```

```
!ubl
```

DLL

Kdexts.dll

Remarks

Here is an example of the use and display of user-space breakpoints:

```
dbgcmd
```

```
kd> !ubp 8014a131
```

```
This command is VERY DANGEROUS, and may crash your system!  
If you don't know what you are doing, enter "!ubc *" now!
```

```
kd> !ubp 801544f4
```

```
kd> !ubd 1
```

```
kd> !ubl
```

```
0: e ffffffff`8014a131 (fffffff`82deb000) 1 ffffffff  
1: d ffffffff`801544f4 (fffffff`82dff000) 0 ffffffff
```

Each line in this listing contains the breakpoint number, the status (**e** for enabled or **d** for disabled), the virtual address used to set the breakpoint, the physical address of the actual breakpoint, the byte position, and the contents of this memory location at the time the breakpoint was set.

See also

[!ubc](#)

[!ubd](#)

!ube

!ubp

User Space and System Space

!ubp

Article • 10/25/2023

The **!ubp** extension sets a breakpoint in user space.



Parameters

Address

Specifies the hexadecimal virtual address of the location in user space where the breakpoint is to be set.

DLL

Kdexts.dll

Remarks

The **!ubp** extension sets a breakpoint in user space. The breakpoint is set on the actual physical page, not just the virtual page.

Setting a physical breakpoint will simultaneously modify every virtual copy of a page, with unpredictable results. One possible consequence is corruption of the system state, possibly followed by a bug check or other system crash. Therefore, these breakpoints should be used cautiously, if at all.

This extension cannot be used to set breakpoints on pages that have been swapped out of memory. If a page is swapped out of memory after a breakpoint is set, the breakpoint ceases to exist.

It is not possible to set a breakpoint inside a page table or a page directory.

Each breakpoint is assigned a *breakpoint number*. To find out the breakpoint number assigned, use [!ubl](#). Breakpoints are enabled upon creation. To step over a breakpoint, you must first disable it by using [!ubd](#). To clear a breakpoint, use [!ubc](#).

See also

!ubc

!ubd

!ube

!ubl

User Space and System Space

!urb

Article • 04/02/2024

The **!urb** extension command is obsolete. Use the [dt URB](#) command instead.

Feedback

Was this page helpful?

 Yes

 No

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

!vad

Article • 04/03/2024

The **!vad** extension displays details of a virtual address descriptor (VAD) or a tree of VADs.

- Displays details of one virtual address descriptor (VAD)
- Displays details of a tree of VADs.
- Displays information about the VADs for a particular user-mode module and provides a string that you can use to load the symbols for that module.

```
dbgcmd
```

```
!vad VAD-Root [Flag]  
!vad Address 1
```

Parameters

VAD-Root

Address of the root of the VAD tree to be displayed.

Flag

Specifies the form the display will take. Possible values include:

0

The entire VAD tree based at *VAD-Root* is displayed. (This is the default.)

1

Only the VAD specified by *VAD-Root* is displayed. The display will include a more detailed analysis.

Address

Address in the virtual address range of a user-mode module.

DLL

Kdexts.dll

Additional Information

For information about virtual address descriptors, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

The address of the root of the VAD for any process can be found by using the [!process](#) command.

The [!vad](#) command can be helpful when you need to load symbols for a user-mode module that has been paged out of memory. For details, see [Mapping Symbols When the PEB is Paged Out](#).

Here is an example of the [!vad](#) extension:

```
dbgcmd

kd> !vad 824bc2f8
VAD      level      start      end      commit
82741bf8 ( 1)    78000     78045      8 Mapped   Exe  EXECUTE_WRITECOPY
824ef368 ( 2)    7f6f0     7f7ef      0 Mapped   EXECUTE_READ
824bc2f8 ( 0)    7ffb0     7ffd3      0 Mapped   READONLY
8273e508 ( 2)    7ffde     7ffde      1 Private  EXECUTE_READWRITE
82643fc8 ( 1)    7ffd0     7ffd0      1 Private  EXECUTE_READWRITE

Total VADs:      5  average level:      2  maximum depth: 2

kd> !vad 824bc2f8 1

VAD @ 824bc2f8
  Start VPN:        7ffb0  End VPN:        7ffd3  Control Area:  827f1208
  First ProtoPte: e1008500  Last PTE e100858c  Commit Charge      0 (0.)
  Secured.Flink      0  Blink          0  Banked/Extend:      0
  Offset 0
    ViewShare NoChange READONLY

  SecNoChange
```

Feedback

Was this page helpful?

 Yes

 No

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

!vad_reload

Article • 04/03/2024

The **!vad_reload** extension reloads the user-mode modules for a specified process by using the virtual address descriptors (VADs) of that process.

```
dbgcmd
```

```
!vad_reload Process
```

Parameters

Process

Specifies the hexadecimal address of the process for which the modules will be loaded.

Additional Information

For information about VADs, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

You can use the [!process](#) extension to find the address of a process.

Here is an example of how to find an address and use it in the **!vad_reload** command.

```
dbgcmd
```

```
3: kd> !process 0 0
...
PROCESS ffffffa8007d54450
    SessionId: 1 Cid: 115c Peb: 7fffffef6000 ParentCid: 0c58
    DirBase: 111bc3000 ObjectTable: ffffff8a006ae0960 HandleCount: 229.
    Image: SearchProtocolHost.exe
...
3: kd> !vad_reload ffffffa8007d54450
fffffa80`04f5e8b0: VAD maps 00000000`6c230000 - 00000000`6c26bfff, file
cscobj.dll
fffffa80`04e8f890: VAD maps 00000000`6ef90000 - 00000000`6f04afff, file
mssvp.dll
fffffa80`07cbb010: VAD maps 00000000`6f910000 - 00000000`6faf5fff, file
tquery.dll
fffffa80`08c1f2a0: VAD maps 00000000`6fb80000 - 00000000`6fb9bfff, file
```

```
mssprxy.dll  
fffffa80`07dce8b0: VAD maps 00000000`6fba0000 - 00000000`6fba7fff, file  
msshooks.dll  
fffffa80`04fd2e70: VAD maps 00000000`72a50000 - 00000000`72a6cff, file  
userenv.dll  
. . .
```

DLL

Kdexts.dll

See also

[!process](#)

[!vad](#)

Feedback

Was this page helpful?

 Yes

 No

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

!validateclist

Article • 04/03/2024

The **!validateclist** extension verifies that the backward and forward links in a doubly-linked list are valid.

```
dbgcmd
```

```
!validateclist Address
```

Parameters

Address

The address of the doubly-linked list.

DLL

Kdexts.dll

Remarks

To stop execution, press Ctrl+Break (in WinDbg) or Ctrl+C (in KD).

Feedback

Was this page helpful?

 Yes

 No

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

!Verifier

Article • 10/25/2023

The **!Verifier** extension displays the status of Driver Verifier and its actions.

Driver Verifier is included in Windows. It works on both checked and free builds. For information about Driver Verifier, see [Driver Verifier](#).

Syntax

```
dbgcmd

!Verifier [Flags [Image]]
!Verifier 4 [Quantity]
!Verifier 8 [Quantity]
!Verifier 0x40 [Quantity]
!Verifier 0x80 [Quantity]
!Verifier 0x80 Address
!Verifier 0x100 [Quantity]
!Verifier 0x100 Address
!Verifier 0x200 [Address]
!Verifier 0x400 [Address]
!Verifier -disable
!Verifier ?
```

Parameters

Flags

Specifies what information is displayed in the output from this command. If *Flags* is equal to the value 4, 8, 0x20, 0x40, 0x80, or 0x100, then the remaining arguments to **!Verifier** are interpreted based on the specific arguments associated with those values. If *Flags* is equal to any other value, even if one or more of these bits are set, only the *Flags* and *Image* arguments are permitted. *Flags* can be any sum of the following bits; the default is 0:

Bit 0 (0x1)

Displays the names of all drivers being verified. The number of bytes currently allocated to each driver from the nonpaged pool and the paged pool is also displayed.

Bit 1 (0x2)

Displays information about pools (pool size, headers, and pool tags) and outstanding memory allocations left by unloaded drivers. This flag has no effect unless bit 0 (0x1) is also set.

Bit 2 (0x4)

Displays fault injection information. The return address, symbol name, and displacement of the code requesting each allocation are displayed. If *Flags* is exactly 0x4 and the *Quantity* parameter is included, the number of these records displayed can be chosen. Otherwise, four records are displayed.

Bit 3 (0x8)

Displays the most recent IRQL changes made by the drivers being verified. The old IRQL, new IRQL, processor, and time stamp are displayed. If *Flags* is exactly 0x8 and the *Quantity* parameter is included, the number of these records displayed can be chosen. Otherwise, four records are displayed.

Warning In 64-bit versions of Windows, some of the kernel functions that raise or lower the IRQL are implemented as inline code rather than as exported functions. Driver Verifier does not report IRQL changes made by inline code, so it is possible for the IRQL transition log produced by Driver Verifier to be incomplete. See Remarks for an example of a missing IRQL transition entry.

Bit 6 (0x40)

(Windows Vista and later) Displays information from the **Force Pending I/O Requests** option of Driver Verifier, including traces from the log of forced pending IRPs.

The *Quantity* parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

Bit 7 (0x80)

(Windows Vista and later) Displays information from the kernel pool Allocate/Free log.

The *Quantity* parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

If *Address* is specified, only traces associated with the specified address within the kernel pool Allocate/Free log are displayed.

Bit 8 (0x100)

(Windows Vista and later) Displays information from the log of *IoAllocateIrp*, *IoCompleteRequest* and *IoCancelIrp* calls.

The *Quantity* parameter specifies the number of traces to be displayed. By default, the entire log is displayed.

If *Address* is specified, only traces associated with the specified IRP address are displayed.

Bit 9 (0x200)

(Windows Vista and later) Displays entries in the Critical Region log.

If *Address* is specified, only entries associated with the specified thread address are displayed.

Bit 10 (0x400)

(Windows Vista and later) Displays cancelled IRPs that are currently being watched by Driver Verifier.

If *Address* is specified, only the IRP with the specified address is displayed.

Bit 11 (0x800)

(Windows 8.1 and later) Display entries from the fault injection log that is created when you select the [Systematic low resource simulation](#) option.

Image

If *Flags* is used and is not equal to 4, 8, or 0x10, *Image* specifies the name of a driver.

Image is used to filter the information displayed by *Flags* values of 0x1 and 0x2: only the specified driver is considered. This driver must be currently verified.

Quantity

If *Flags* is exactly equal to 0x4, *Quantity* specifies the number of fault injection records to display. If *Flags* is exactly equal to 0x8, *Quantity* specifies the number of IRQL log entries to display. If *Flags* is exactly equal to 0x40, *Quantity* specifies the number of traces displayed from the log of forced pending IRPs. If *Flags* is exactly equal to 0x80, *Quantity* specifies the number of traces displayed from the kernel pool Allocate/Free log. If *Flags* is exactly equal to 0x100, *Quantity* specifies the number of traces displayed from the log of IoAllocateIrp, IoCompleteRequest and IoCancelIrp calls.

-disable

Clears the current Driver Verifier settings on the debug target. The clearing of these settings does not persist through a reboot. If you need to disable the Driver Verifier settings to successfully boot, set a breakpoint at nt!VerifierInitSystem and use the !Verifier -disable command at that point.

?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Kdexts.dll

Additional Information

For information about [Driver Verifier](#), see the Windows Driver Kit (WDK) documentation.

Remarks

The following example illustrates that on 64-bit versions of Windows, the IRQL transition log is not always complete. The two entries shown are consecutive entries in the log for Processor 2. The first entry shows the IRQL going from 2 to 0. The second entry shows the IRQL going from 2 to 2. Information about how the IRQL got raised from 0 to 2 is missing.

```
dbgcmd

Thread:          ffffffa80068c9400
Old irql:        0000000000000002
New irql:        0000000000000000
Processor:       0000000000000002
Time stamp:     0000000000000857

fffff8800140f12a ndis!ndisNsiGetInterfaceInformation+0x20a
fffff88001509478 NETIO!NsiGetParameterEx+0x178
fffff88005f062f2 nsiproxy!NsippGetParameter+0x24a
fffff88005f086db nsiproxy!NsippDispatchDeviceControl+0xa3
fffff88005f087a0 nsiproxy!NsippDispatch+0x48

Thread:          ffffffa80068c9400
Old irql:        0000000000000002
New irql:        0000000000000002
Processor:       0000000000000002
Time stamp:     0000000000000857

fffff8800140d48d ndis!ndisReferenceTopMiniportByNameForNsi+0x1ce
fffff8800140f072 ndis!ndisNsiGetInterfaceInformation+0x152
fffff88001509478 NETIO!NsiGetParameterEx+0x178
fffff88005f062f2 nsiproxy!NsippGetParameter+0x24a
fffff88005f086db nsiproxy!NsippDispatchDeviceControl+0xa3
```

The values of 4, 8, and 0x20, 0x40, 0x80, and 0x100 are special values for *Flags*. If these values are used, the special arguments listed in the **Parameters** section can be used, and the display will include only the information associated with that flag value.

If any other value for *Flags* is used, even if one or more of these bits are set, only the *Flags* and *Image* arguments are permitted. In this situation, in addition to all the other information displayed, **!Verifier** will display the Driver Verifier options that are active, along with statistics on pool allocations, IRQL raises, spin locks, and trims.

If *Flags* equals 0x20, the values specified for *CompletionTime*, *CancelTime*, and *ForceCancellation* are used by the Driver Hang Verification option of Driver Verifier. These new values take effect immediately and last until the next boot. When you reboot, they revert to their default values.

Also, if *Flags* equals 0x20 (with or without additional parameters), the Driver Hang Verification log is printed. For information on interpreting the log, see the Driver Hang Verification section of the Driver Verifier documentation in the Windows Driver Kit (WDK) documentation.

Here is an example of the !verifier extension on a Windows 7 computer.

```
dbgcmd

2: kd> !Verifier 0xf

Verify Level 9bb ... enabled options are:
    Special pool
    Special irql
    All pool allocations checked on unload
    Io subsystem checking enabled
    Deadlock detection enabled
    DMA checking enabled
    Security checks enabled
    Miscellaneous checks enabled

Summary of All Verifier Statistics

RaiseIrqls                      0x0
AcquireSpinLocks                 0x362
Synch Executions                  0x0
Trims                           0xa34a

Pool Allocations Attempted       0x7b058
Pool Allocations Succeeded       0x7b058
Pool Allocations Succeeded SpecialPool 0x7b058
Pool Allocations With NO TAG     0x0
Pool Allocations Failed          0x0
Resource Allocations Failed Deliberately 0x0

Current paged pool allocations   0x1a for 00000950 bytes
Peak paged pool allocations      0x1b for 00000AC4 bytes
Current nonpaged pool allocations 0xe3 for 00046110 bytes
Peak nonpaged pool allocations    0x10f for 00048E40 bytes

Driver Verification List

Entry      State           NonPagedPool    PagedPool    Module
fffffa8003b6f670 Loaded        000000a0        00000854    videoprpt.sys

Current Pool Allocations 00000002    00000013
```

```
Current Pool Bytes      000000a0      00000854
Peak Pool Allocations   00000006      00000014
Peak Pool Bytes         000008c0      000009c8

PoolAddress  SizeInBytes  Tag          CallersAddress
fffff9800157efc0      0x0000003c  Vprt        fffff88002c62963
fffff9800146afc0      0x00000034  Vprt        fffff88002c62963
fffff980015bafe0      0x00000018  Vprt        fffff88002c628f7
...
fffffa8003b6f620 Loaded           00046070      000000fc      usbport.sys
```

```
Current Pool Allocations 000000e1      00000007
Current Pool Bytes       00046070      000000fc
Peak Pool Allocations    0000010d      0000000a
Peak Pool Bytes          00048da0      00000254
```

```
PoolAddress  SizeInBytes  Tag          CallersAddress
fffff98003a38fc0      0x00000038  usbp        fffff88004215e34
fffff98003a2cf0       0x00000038  usbp        fffff88004215e34
fffff9800415efc0      0x00000038  usbp        fffff88004215e34
...
-----
```

```
Fault injection trace log
```

```
Driver Verifier didn't inject any faults.
```

```
Track irql trace log
```

```
Displaying most recent 0x0000000000000004 entries from the IRQL transition
log.
```

```
There are up to 0x100 entries in the log.
```

```
Thread:              fffff80002bf8c40
Old irql:            0000000000000002
New irql:            0000000000000002
Processor:          0000000000000000
Time stamp:          00000000000495e
```

```
fffff8800420f2ca USBPORT!USBPORT_DM_IoTimerDpc+0x9a
fffff80002a5b5bf nt!IopTimerDispatch+0x132
fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6
fffff80002a7c4be nt!KiTimerExpiration+0x1be
```

```
Thread:              fffff80002bf8c40
Old irql:            0000000000000002
New irql:            0000000000000002
Processor:          0000000000000000
Time stamp:          00000000000495e
```

```
fffff88004205f3a USBPORT!USBPORT_AcquireEpListLock+0x2e
fffff880042172df USBPORT!USBPORT_Core_TimeoutAllTransfers+0x1f
fffff8800420f2ca USBPORT!USBPORT_DM_IoTimerDpc+0x9a
fffff80002a5b5bf nt!IopTimerDispatch+0x132
fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
```

```
Thread:          fffff80002bf8c40
Old irql:        0000000000000002
New irql:        0000000000000002
Processor:       0000000000000000
Time stamp:     00000000000495e
```

```
fffff88004201694 USBPORT!MPf_CheckController+0x4c
fffff8800420f26a USBPORT!USBPORT_DM_IoTimerDpc+0x3a
fffff80002a5b5bf nt!IopTimerDispatch+0x132
fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6
```

```
Thread:          fffff80002bf8c40
Old irql:        0000000000000002
New irql:        0000000000000002
Processor:       0000000000000000
Time stamp:     00000000000495e
```

```
fffff8800420167c USBPORT!MPf_CheckController+0x34
fffff8800420f26a USBPORT!USBPORT_DM_IoTimerDpc+0x3a
fffff80002a5b5bf nt!IopTimerDispatch+0x132
fffff80002a7c29e nt!KiProcessTimerDpcTable+0x66
fffff80002a7bdd6 nt!KiProcessExpiredTimerList+0xc6
```

Here is an example of the !verifier extension on a Windows Vista computer with bit 7 turned on and *Address* specified.

```
dbgcmd

0: kd> !Verifier 80 a2b1cf20
# Parsing 00004000 array entries, searching for address a2b1cf20.

Pool block a2b1ce98, Size 00000168, Thread a2b1ce98
808f1be6 ndis!ndisFreeToNPagedPool+0x39
808f11c1 ndis!ndisPplFree+0x47
808f100f ndis!NdisFreeNetBufferList+0x3b
8088db41 NETIO!NetioFreeNetBufferAndNetBufferList+0xe
8c588d68 tcpip!UdpEndSendMessages+0xdf
8c588cb5 tcpip!UdpSendMessagesDatagramsComplete+0x22
8088d622 NETIO!NetioDereferenceNetBufferListChain+0xcf
8c5954ea tcpip!FlSendNetBufferListChainComplete+0x1c
809b2370 ndis!ndisMSendCompleteNetBufferListsInternal+0x67
808f1781 ndis!NdisFSendNetBufferListsComplete+0x1a
8c04c68e pacer!PcFilterSendNetBufferListsComplete+0xb2
809b230c ndis!ndisMSendNetBufferListsComplete+0x70
# 8ac4a8ba test1!HandleCompletedTxPacket+0xea
```

```
Pool block a2b1ce98, Size 00000164, Thread a2b1ce98
822af87f nt!VerifierExAllocatePoolWithTagPriority+0x5d
808f1c88 ndis!ndisAllocateFromNPagedPool+0x1d
808f11f3 ndis!ndisPplAllocate+0x60
808f1257 ndis!NdisAllocateNetBufferList+0x26
80890933
NETIO!NetioAllocateAndReferenceNetBufferListNetBufferMdlAndData+0x14
8c5889c2 tcpip!UdpSendMessages+0x503
8c05c565 afd!AfdTLSendMessages+0x27
8c07a087 afd!AfdTLFastDgramSend+0x7d
8c079f82 afd!AfdFastDatagramSend+0x5ae
8c06f3ea afd!AfdFastIoDeviceControl+0x3c1
8217474f nt!IopXxxControlFile+0x268
821797a1 nt!NtDeviceIoControlFile+0x2a
8204d16a nt!KiFastCallEntry+0x127
```

!vm

Article • 10/25/2023

The **!vm** extension displays summary information about virtual memory use statistics on the target system.

dbgcmd

```
!vm [Flags]
```

Parameters

Flags

Specifies what information will be displayed in the output from this command. This can be any sum of the following bits. The default is 0, which causes the display to include system-wide virtual memory statistics as well as memory statistics for each process.

Bit 0 (0x1)

Causes the display to omit process-specific statistics.

Bit 1 (0x2)

Causes the display to include memory management thread stacks.

Bit 2 (0x4)

Causes the display to include terminal server memory usage.

Bit 3 (0x8)

Causes the display to include the page file write log.

Bit 4 (0x10)

Causes the display to include working set owner thread stacks.

Bit 5 (0x20)

(Windows Vista and later) Causes the display to include kernel virtual address usage.

Environment

Modes: kernel mode only

DLL

Additional Information

The [!memusage](#) extension command can be used to analyze physical memory usage. For more information about memory management, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example of the short output produced when *Flags* is 1:

```
dbgcmd

kd> !vm 1

*** Virtual Memory Usage ***
Physical Memory:    16270  (   65080 Kb)
Page File: \??\E:\pagefile.sys
    Current:    98304Kb Free Space:    61044Kb
Minimum:    98304Kb Maximum:    196608Kb
    Available Pages:    5543  (   22172 Kb)
    ResAvail Pages:    6759  (   27036 Kb)
    Locked IO Pages:    112  (     448 Kb)
Free System PTEs:    45089  ( 180356 Kb)
    Free NP PTEs:    5145  (   20580 Kb)
    Free Special NP:    336  (     1344 Kb)
    Modified Pages:    714  (     2856 Kb)
    NonPagedPool Usage:    877  (     3508 Kb)
    NonPagedPool Max:    6252  (   25008 Kb)
    PagedPool 0 Usage:    729  (     2916 Kb)
    PagedPool 1 Usage:    432  (     1728 Kb)
    PagedPool 2 Usage:    436  (     1744 Kb)
    PagedPool Usage:    1597  (     6388 Kb)
    PagedPool Maximum:    13312  (   53248 Kb)
    Shared Commit:    1097  (     4388 Kb)
    Special Pool:    229  (     916 Kb)
    Shared Process:    1956  (     7824 Kb)
    PagedPool Commit:    1597  (     6388 Kb)
    Driver Commit:    828  (     3312 Kb)
    Committed pages:    21949  (   87796 Kb)
    Commit limit:    36256  ( 145024 Kb)
```

All memory use is listed in pages and in kilobytes. The most useful information in this display is the following:

Parameter	Meaning
physical memory	Total physical memory in the system.
available pages	Number of pages of memory available on the system, both virtual and physical.
nonpaged pool usage	The amount of pages allocated to the nonpaged pool. The nonpaged pool is memory that cannot be swapped out to the paging file, so it must always occupy physical memory. If this number is too large, this is usually an indication that there is a memory leak somewhere in the system.

!vpb

Article • 10/25/2023

The **!vpb** extension displays a volume parameter block (VPB).

```
dbgcmd
```

```
!vpb Address
```

Parameters

Address

Specifies the hexadecimal address of the VPB.

DLL

Kdexts.dll

Additional Information

For information about VPBs, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Here is an example. First, the device tree is displayed with the [!devnode](#) extension:

```
dbgcmd
```

```
kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x80e203b8)
DevNode 0x80e203b8 for PDO 0x80e204f8
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x80e56dc8 for PDO 0x80e56f18
    InstancePath is "Root\dmio\0000"
    ServiceName is "dmio"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x80e56ae8 for PDO 0x80e56c38
    InstancePath is "Root\ftdisk\0000"
    ServiceName is "ftdisk"
```

```
State = DeviceNodeStarted (0x308)
Previous State = DeviceNodeEnumerateCompletion (0x30d)
DevNode 0x80e152a0 for PDO 0x80e15cb8
    InstancePath is
"STORAGE\Volume\1&30a96598&0&Signature5C34D70COffset7E00Length60170A00"
    ServiceName is "VolSnap"
    TargetDeviceNotify List - f 0xe1250938 b 0xe14b9198
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
....
```

The last device node listed is a volume. Examine its physical device object (PDO) with the [!devobj](#) extension:

```
dbgcmd

kd> !devobj 80e15cb8
Device object (80e15cb8) is for:
    HarddiskVolume1 \Driver\Ftdisk DriverObject 80e4e248
    Current Irp 00000000 RefCount 14 Type 00000007 Flags 00001050
    Vpb 80e15c30 DevExt 80e15d70 DevObjExt 80e15e40 Dope 80e15bd8 DevNode
    80e152a0
    ExtensionFlags (0000000000)
    AttachedDevice (Upper) 80e14c60 \Driver\VolSnap
    Device queue is not busy.
```

The address of this device's VPB is included in this listing. Use this address with the [!vpb](#) extension:

```
dbgcmd

kd> !vpb 80e15c30
Vpb at 0x80e15c30
Flags: 0x1 mounted
DeviceObject: 0x80de5020
RealDevice: 0x80e15cb8
RefCount: 14
Volume Label: MY-DISK-C
```

!vtop

Article • 10/25/2023

The **!vtop** extension converts a virtual address to the corresponding physical address, and displays other page table and page directory information.

Syntax

```
dbgcmd  
!vtop PFN VirtualAddress  
!vtop 0 VirtualAddress
```

Parameters

DirBase

Specifies the directory base for the process. Each process has its own virtual address space. Use the [!process](#) extension to determine the directory base for a process.

PFN

Specifies the page frame number (PFN) of the directory base for the process.

0

Causes **!vtop** to use the current [process context](#) for address translation.

VirtualAddress

Specifies the virtual address whose page is desired.

DLL

Kdexts.dll

Additional Information

For other methods of achieving these results, see [Converting Virtual Addresses to Physical Addresses](#). Also see [!ptov](#). For information about page tables and page directories, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

To use this command, first use the **!process** extension to determine the directory base of the process. The page frame number (PFN) of this directory base can be found by removing the three trailing hexadecimal zeros (in other words, by right-shifting the number 12 bits).

Here is an example:

```
dbgcmd

kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
....
PROCESS ff779190 SessionId: 0 Cid: 04fc Peb: 7ffdf000 ParentCid: 0394
DirBase: 098fd000 ObjectTable: e1646b30 TableSize: 8.
Image: MyApp.exe
```

Since the directory base is 0x098FD000, its PFN is 0x098FD.

```
dbgcmd

kd> !vtop 98fd 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)
```

Notice how the trailing three zeros are optional. The **!vtop** extension displays the page directory index (PDI), the page table index (PTI), the virtual address that you had originally input, the physical address of the beginning of the physical page, and the page frame number (PFN) of the page table entry (PTE).

If you want to convert the virtual address 0x0012F980 to a physical address, you simply have to take the last three hexadecimal digits (0x980) and add them to the physical address of the beginning of the page (0x09DE9000). This gives the physical address 0x09DE9980.

If you forget to remove the three zeros, and pass the full directory base to **!vtop** instead of the PFN, the results will usually be correct. This is because when **!vtop** receives a number too large to be a PFN, it right-shifts it twelve bits and uses that number instead:

```
dbgcmd

kd> !vtop 98fd 12f980
Pdi 0 Pti 12f
0012f980 09de9000 pfn(09de9)

kd> !vtop 98fd000 12f980
```

```
Pdi 0 Pti 12f  
0012f980 09de9000 pfn(09de9)
```

However, it is better to always use the PFN, because some directory base values will not be converted in this manner.

!wdmaud

Article • 04/03/2024

!wdmaud is obsolete.

dbgcmd

!wdmaud Address Flags

Parameters

Address

Specifies the address of the structure to be displayed.

Flags

Specifies the information to display. This must include exactly one of the bits 0x1, 0x2, 0x4, and 0x8. The 0x100 bit can be added to any of these.

Bit 0 (0x1)

Displays a list of all IOCTLs that have been sent to wdmaud.sys. When this is used, *Address* should specify the address of the **WdmalocHistoryListHead**. If the 0x100 bit is set, the display also includes the **pContext** that each IOCTL was sent with.

Bit 1 (0x2)

Displays a list of all IRPs that WDMAud has marked as pending. When this is used, *Address* should specify the address of the **WdmaPendingIrpListHead**. If the 0x100 bit is set, the display also includes the context on which each IRP was allocated.

Bit 2 (0x4)

Displays a list of all MDLs that WDMAud has allocated. When this is used, *Address* should specify the address of the **WdmaAllocatedMdlListHead**. If the 0x100 bit is set, the display also includes the context on which each MDL was allocated.

Bit 3 (0x8)

Displays a list of all active contexts attached to wdmaud.sys. When this is used, *Address* should specify the address of the **WdmaContextListHead**. If the 0x100 bit is set, the display also includes the data members of each context structure.

Bit 8 (0x100)

Causes the display to include verbose information.

DLL

Windows XP and later - Unavailable

Additional Information

For information about WDM audio architecture and audio drivers, see the Windows Driver Kit (WDK) documentation.

Remarks

The contexts attached to wdmaud.sys (**pContext**) contain most of the state data for each device. Whenever wdmaud.drv is loaded into a new process, wdmaud.sys is notified of its arrival. Whenever wdmaud.drv is unloaded, wdmaud.sys cleans up any allocations made in that context.

Feedback

Was this page helpful?



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

!whattime

Article • 04/03/2024

The **!whattime** extension converts a tick count into a standard time value.

```
dbgcmd
```

```
!whattime Ticks
```

Parameters

Ticks

The number of ticks.

DLL

Kdexts.dll

Remarks

The output is displayed as *HH:MM:SS.mmm*. Here is an example:

```
dbgcmd
```

```
kd> !whattime 29857ae4
696613604 Ticks in Standard Time: 15:02:16.040s
```

Feedback

Was this page helpful?

 Yes

 No

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

!whatperftime

Article • 04/03/2024

The `!whatperftime` extension converts a high-resolution performance counter value into a standard time value.

```
dbgcmd
```

```
!whatperftime Count
```

Parameters

Count

The performance counter clock value.

DLL

Kdexts.dll

Remarks

You can use `!whatperftime` to convert values retrieved by calling `QueryPerformanceCounter`. Performance counter time values are also found in software traces.

The output is displayed as `HH:MM:SS.mmm`. Here is an example:

```
dbgcmd
```

```
kd> !whatperftime 304589
3163529 Performance Counter in Standard Time: .004.313s
```

Feedback

Was this page helpful?

 Yes

 No

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

!whea

Article • 04/03/2024

The **!whea** extension displays top-level Windows Hardware Error Architecture (WHEA) information.

```
dbgcmd
```

```
!whea
```

DLL

Windows Vista and later - Kdexts.dll

Additional Information

The **!errrec** and **!errpkt** extensions can be used to display additional WHEA information. For general information about WHEA, see [Windows Hardware Error Architecture \(WHEA\)](#) in the Windows Driver Kit (WDK) documentation.

Remarks

The following example shows the (truncated) output of the **!whea** extension:

```
dbgcmd
```

```
3: kd> !whea
Error Source Table @ ffffff800019ca250
13 Error Sources
Error Source 0 @ ffffffa80064132c0
    Notify Type      : Machine Check Exception
    Type             : 0x0 (MCE)
    Error Count     : 8
    Record Count    : 10
    Record Length   : c3e
    Error Records   :
                      : wrapper @ ffffffa8007cf4000 record @ ffffffa8007cf4030
                      : wrapper @ ffffffa8007cf4c3e record @ ffffffa8007cf4c6e
                      : wrapper @ ffffffa8007cf587c record @ ffffffa8007cf58ac
                      : wrapper @ ffffffa8007cf64ba record @ ffffffa8007cf64ea
                      : wrapper @ ffffffa8007cf70f8 record @ ffffffa8007cf7128
                      : wrapper @ ffffffa8007cf7d36 record @ ffffffa8007cf7d66
                      : wrapper @ ffffffa8007cf8974 record @ ffffffa8007cf89a4
                      : wrapper @ ffffffa8007cf95b2 record @ ffffffa8007cf95e2
                      : wrapper @ ffffffa8007cfa1f0 record @ ffffffa8007cfa220
```

```
: wrapper @ ffffffa8007cfae2e record @ ffffffa8007cfae5e
: wrapper @ ffffffa8007cfba6c record @ ffffffa8007cfba9c
: wrapper @ ffffffa8007cfc6aa record @ ffffffa8007cfc6da
: wrapper @ ffffffa8007cfd2e8 record @ ffffffa8007cfd318
: wrapper @ ffffffa8007cfdf26 record @ ffffffa8007cfdf56
: wrapper @ ffffffa8007cfb64 record @ ffffffa8007cfb94
: wrapper @ ffffffa8007cff7a2 record @ ffffffa8007cff7d2
Descriptor : @ ffffffa8006413328
Length : 3cc
Max Raw Data Length : 392
Num Records To Preallocate : 10
Max Sections Per Record : 3
Error Source ID : 0
Flags : 00000000
Error Source 1 @ ffffffa8007d00bc0
Notify Type : Corrected Machine Check
Type : 0x1 (CMC)
Error Count : 0
Record Count : 10
Record Length : c3e
Error Records : wrapper @ ffffffa8007d01000 record @ ffffffa8007d01030
                : wrapper @ ffffffa8007d01c3e record @ ffffffa8007d01c6e
                : wrapper @ ffffffa8007d0287c record @ ffffffa8007d028ac
                : wrapper @ ffffffa8007d034ba record @ ffffffa8007d034ea
                : wrapper @ ffffffa8007d040f8 record @ ffffffa8007d04128
                : wrapper @ ffffffa8007d04d36 record @ ffffffa8007d04d66
                : wrapper @ ffffffa8007d05974 record @ ffffffa8007d059a4
                : wrapper @ ffffffa8007d065b2 record @ ffffffa8007d065e2
                : wrapper @ ffffffa8007d071f0 record @ ffffffa8007d07220
                : wrapper @ ffffffa8007d07e2e record @ ffffffa8007d07e5e
                : wrapper @ ffffffa8007d08a6c record @ ffffffa8007d08a9c
                : wrapper @ ffffffa8007d096aa record @ ffffffa8007d096da
                : wrapper @ ffffffa8007d0a2e8 record @ ffffffa8007d0a318
                : wrapper @ ffffffa8007d0af26 record @ ffffffa8007d0af56
                : wrapper @ ffffffa8007d0bb64 record @ ffffffa8007d0bb94
                : wrapper @ ffffffa8007d0c7a2 record @ ffffffa8007d0c7d2
Descriptor : @ ffffffa8007d00c28
Length : 3cc
Max Raw Data Length : 392
Num Records To Preallocate : 10
Max Sections Per Record : 3
Error Source ID : 1
Flags : 00000000
Error Source 2 @ ffffffa8007d00770
Notify Type : Non-Maskable Interrupt
Type : 0x3 (NMI)
Error Count : 1
Record Count : 1
Record Length : 1f8
Error Records : wrapper @ ffffffa800631b2c0 record @ ffffffa800631b2f0
Descriptor : @ ffffffa8007d007d8
Length : 3cc
Max Raw Data Length : 100
Num Records To Preallocate : 1
Max Sections Per Record : 1
```

```
Error Source ID          : 2
Flags                  : 00000000
Error Source 3 @ ffffffa8007d0dbc0
  Notify Type      : BOOT Error Record
  Type            : 0x7 (BOOT)
  Error Count     : 0
  Record Count    : 1
  Record Length   : 4f8
  Error Records   : wrapper @ 0000000000000000 record @ 000000000000030
  Descriptor      : @ ffffffa8007d0dc28
    Length         : 3cc
    Max Raw Data Length : 400
    Num Records To Preallocate : 1
    Max Sections Per Record : 1
    Error Source ID       : 3
    Flags              : 00000000
.
.
```

Feedback

Was this page helpful?

 Yes

 No

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

!wsle

Article • 04/03/2024

The **!wsle** extension displays all working set list entries (WSLEs).

Syntax

```
dbgcmd
```

```
!wsle [Flags [Address]]
```

Parameters

Flags

Specifies the information to include in the display. This can be any combination of the following bits. The default is zero. If this is used, only basic information about the working set is displayed.

Bit 0 (0x1)

Causes the display to include information about each WSLE's address, age, lock status, and reference count. If a WSLE has an invalid page table entry (PTE) or page directory entry (PDE) associated with it, this is also displayed.

Bit 1 (0x2)

Causes the display to include the total number of valid WSLEs, the index of the last WSLE, and the index of the first free WSLE.

Bit 2 (0x4)

Causes the display to include the total number of free WSLEs, as well as the index of each free WSLE. If bit 1 is also set, then a check is done to make sure that the number of free WSLEs plus the number of valid WSLEs is actually equal to the total number of WSLEs.

Address

Specifies the address of the working set list. If this is omitted, the default working set list is used. Specifying zero for *Address* is the same as omitting it.

DLL

Kdexts.dll

Additional Information

For information about working sets, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

This extension can take a significant amount of time to execute.

Here is an example from an x86 target computer running Windows Server 2003:

```
dbgcmd

kd> !wsle 3

Working Set @ c0503000
FirstFree:      a7  FirstDynamic:          4
LastEntry       23d  NextSlot:             4  LastInitialized:    259
NonDirect       65   HashTable:            0  HashTableSize:     0

Reading the WSLE data...

Virtual Address        Age  Locked  ReferenceCount
c0300203              0    1        1
c0301203              0    1        1
c0502203              0    1        1
c0503203              0    1        1
c01ff201              0    0        1
77f74d19              3    0        1
7ffdffa01             2    0        1
c0001201              0    0        1

.....
Reading the WSLE data...
Valid WSLE entries = 0xa7
found end @ wsle index 0x259

.....
```

Feedback

Was this page helpful?

 Yes

 No

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

!zombies

Article • 04/03/2024

The **!zombies** extension displays all dead ("zombie") processes or threads.

```
dbgcmd
```

```
!zombies [Flags [RestartAddress]]
```

Parameters

Flags

Specifies what will be displayed. Possible values include:

1

Displays all zombie processes. (This is the default.)

2

Displays all zombie threads.

RestartAddress

Specifies the hexadecimal address at which to begin the search. This is useful if the previous search was terminated prematurely. The default is zero.

DLL

Windows XP and later - Unavailable

Additional Information

This extension is obsolete.

To see a list of all processes and threads, use the [!process](#) extension.

For general information about processes and threads in kernel mode, see [Changing Contexts](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals*, by Mark Russinovich and David Solomon.

Remarks

Zombie processes are dead processes that have not yet been removed from the process list. Zombie threads are analogous.

Feedback

Was this page helpful?

 Yes

 No

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

User-Mode Extensions

Article • 10/25/2023

This section of the reference describes extension commands that are primarily used during user-mode debugging.

The debugger will automatically load the proper version of these extension commands. Unless you have manually loaded a different version, you do not have to keep track of the DLL versions being used. See [Using Debugger Extension Commands](#) for a description of the default module search order. See [Loading Debugger Extension DLLs](#) for an explanation of how to load extension modules.

Each extension command reference page lists the DLLs that expose that command. Use the following rules to determine the proper directory from which to load this extension DLL:

- If your target application is running on Windows XP or a later version of Windows, use `winxp\Ntsdexts.dll`.

In addition, user-mode extensions that are not specific to any single operating system can be found in `winext\Uext.dll`.

!avr

Article • 10/25/2023

The **!avr** extension controls the settings of [Application Verifier](#) and displays a variety of output produced by Application Verifier.

dbgcmd

```
!avr
!avr -vs { Length | -a Address }
!avr -hp { Length | -a Address }
!avr -cs { Length | -a Address }
!avr -dlls [ Length ]
!avr -trm
!avr -ex [ Length ]
!avr -threads [ ThreadID ]
!avr -tp [ ThreadID ]
!avr -srw [ Address | Address Length ] [ -stats ]
!avr -leak [ -m ModuleName ] [ -r ResourceType ] [ -a Address ] [ -t ]
!avr -trace TraceIndex
!avr -cnt
!avr -brk [BreakEventType]
!avr -flt [EventType Probability]
!avr -flt break EventType
!avr -flt stacks Length
!avr -trg [ Start End | dll Module | all ]
!avr -settings
!avr -skp [ Start End | dll Module | all | Time ]
```

Parameters

-vs { Length | -a Address }

Displays the virtual space operation log. *Length* specifies the number of records to display, starting with the most recent. *Address* specifies the virtual address. Records of the virtual operations that contain this virtual address are displayed.

-hp { Length | -a Address }

Displays the heap operation log. *Address* specifies the heap address. Records of the heap operations that contain this heap address are displayed.

-cs { Length | -a Address }

Displays the critical section delete log. *Length* specifies the number of records to display, starting with the most recent. *Address* specifies the critical section address. Records for the particular critical section are displayed when *Address* is specified.

-dlls [*Length*]

Displays the DLL load/unload log. *Length* specifies the number of records to display, starting with the most recent.

-trm

Displays a log of all terminated and suspended threads.

-ex [*Length*]

Displays the exception log. Application Verifier tracks all the exceptions in the application.

-threads [*ThreadID*]

Displays information about threads in the target process. For child threads, the stack size and the **CreateThread** flags specified by the parent are also displayed. If you provide a thread ID, information for only that thread is displayed.

-tp [*ThreadID*]

Displays the threadpool log. This log contains stack traces for various operations such as changing the thread affinity mask, changing thread priority, posting thread messages, and initializing or uninitialized COM from within the threadpool callback. If you provide a thread ID, information for that thread only is displayed.

-srw [*Address* | *Address Length*] [-stats]

Displays the Slim Reader/Writer (SRW) log. If you specify *Address*, records for the SRW lock at that address are displayed. If you specify *Address* and *Length*, records for SRW locks in that address range are displayed. If you include the **-stats** option, the SRW lock statistics are displayed.

-leak [-m *ModuleName*] [-r *ResourceType*] [-a *Address*] [-t]

Displays the outstanding resources log. These resources may or may not be leaks at any given point. If you specify *Modulename* (including the extension), all outstanding resources in the specified module are displayed. If you specify *ResourceType*, all outstanding resources of that resource type are displayed. If you specify *Address*, records of outstanding resources with that address are displayed. *ResourceType* can be one of the following:

Heap: Displays heap allocations using Win32 Heap APIs

Local: Displays Local/Global allocations

CRT: Displays allocations using CRT APIs

Virtual: Displays Virtual reservations

BSTR: Displays BSTR allocations

Registry: Displays Registry key opens

Power: Displays power notification objects

Handle: Displays thread, file, and event handle allocations

-trace *TraceIndex* Displays a stack trace for the specified trace index. Some structures use this 16-bit index number to identify a stack trace. This index points to a location within the stack trace database.

-cnt Displays a list of global counters.

-brk [*BreakEventType*] Specifies a break event. *BreakEventType* is the type number of the break event. For a list of possible types, and a list of the current break event settings, enter !avrf -brk.

-flt [*EventType Probability*] Specifies a fault injection. *EventType* is the type number of the event. *Probability* is the frequency with which the event will fail. This can be any integer between 0 and 1,000,000 (0xF4240). If you enter !avrf -flt with no additional parameters, the current fault injection settings are displayed.

-flt break *EventType* Causes Application Verifier to break into the debugger each time this fault, specified by *EventType*, is injected.

-flt stacks *Length* Displays *Length* number of stack traces for the most recent fault-injected operations.

-trg [*Start End | dll Module | all*] Specifies a target range. *Start* is the beginning address of the target range. *End* is the ending address of the target range. *Module* specifies the name (including the .exe or .dll extension, but not including the path) of a module to be targeted. If you enter -trg all, all target ranges are reset. If you enter -trg with no additional parameters, the current target ranges are displayed.

-skp [*Start End | dll Module | all | Time*] Specifies an exclusion range. *Start* is the beginning address of the exclusion range. *End* is the ending address of the exclusion range. *Module* specifies the name of a module to be targeted or excluded. *Module* specifies the name (including the .exe or .dll extension, but not including the path) of a module to be excluded. If you enter -skp all, all target ranges or exclusion ranges are reset. If you enter a *Time* value, all faults are suppressed for *Time* milliseconds after execution resumes.

DLL

exts.dll

Additional Information

For information about how to download and install Application Verifier and its documentation, see [Application Verifier](#).

Remarks

When the **!avr** extension is used with no parameters, it displays the current Application Verifier options. If the **Full page heap** or **Fast fill heap** option has been enabled, information about active page heaps is displayed as well. For some examples, see "Heap Operation Logs" in [Debugging Application Verifier Stops](#).

If an Application Verifier Stop has occurred, the **!avr** extension with no parameters will reveal the nature of the stop and its cause. For some examples, see [Debugging Application Verifier Stops](#).

If symbols for ntdll.dll and verifier.dll are missing, the **!avr** extension generates an error message. For information about how to address this problem, see "Debugger install and setup" in [Debugging Application Verifier Stops](#).

!critsec

Article • 10/25/2023

The **!critsec** extension displays a critical section.

dbgsyntax

!critsec Address

Parameters

Address

Specifies the hexadecimal address of the critical section.

DLL

Ntsdexts.dll

Additional Information

For other commands and extensions that can display critical section information, see [Displaying a Critical Section](#). For information about critical sections, see the Microsoft Windows SDK documentation, the Windows Driver Kit (WDK) documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

If you do not know the address of the critical section, you should use the [!ntsdexts.locks](#) extension. This displays all critical sections that have been initialized by calling `RtlInitializeCriticalSection`.

Here is an example:

dbgcmd

```
0:000> !critsec 3a8c0e9c

CritSec +3a8c0e9c at 3A8C0E9C
LockCount      1
RecursionCount 1
```

OwningThread	99
EntryCount	472
ContentionCount	1
*** Locked	

See also

[Displaying a Critical Section](#)

[CriticalSection Time Outs \(user mode\)](#)

[!cs](#)

!dp (!ntsdexts.dp)

Article • 04/03/2024

The **!dp** extension in Ntsdexts.dll displays a CSR process.

This extension command should not be confused with the [dp \(Display Memory\)](#) command, or with the [!kdext*.dp](#) extension command.

dbgcmd

```
!dp [v] [ PID | CSR-Process ]
```

Parameters

v

Verbose mode. Causes the display to include structure and thread list.

PID

Specifies the process ID of the CSR process.

CSR-Process

Specifies the hexadecimal address of the CSR process.

DLL

Ntsdexts.dll

Remarks

This extension displays the process address, process ID, sequence number, flags, and reference count. If verbose mode is selected, additional details are displayed, and thread information is shown for each process.

If no process is specified, all processes are displayed.

See also

[!dt](#)

Feedback

Was this page helpful?

 Yes

 No

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

!dreg

Article • 04/03/2024

The **!dreg** extension displays registry information.

dbgcmd

```
!dreg [-d|-w] KeyPath[!Value]  
!dreg
```

Parameters

-d

Causes binary values to be displayed as DWORDs.

-w

Causes binary values to be displayed as WORDs.

KeyPath

Specifies the registry key path. It can begin with any of the following abbreviations:

hklm

HKEY_LOCAL_MACHINE

hkcu

HKEY_CURRENT_USER

hkcr

HKEY_CLASSES_ROOT

hku

HKEY_USERS

If no abbreviation is used, HKEY_LOCAL_MACHINE is assumed.

Value

Specifies the name of the registry value to be displayed. If an asterisk (*) is used, all values are displayed. If *Value* is omitted, all subkeys are displayed.

DLL

Ntsdexts.dll

Additional Information

For information about the registry, see the Windows Driver Kit (WDK) documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!dreg** extension can be used to display the registry during user-mode debugging.

It is most useful during remote debugging, as it allows you to browse the registry of the remote machine. It is also useful when controlling the user-mode debugger from the kernel debugger, because you cannot run a standard registry editor on the target machine when it is frozen. (You can use the [.sleep](#) command for this purpose as well. See [Controlling the User-Mode Debugger from the Kernel Debugger](#) for details.)

It is also useful when debugging locally, as the information is presented in an easily readable format.

If **!dreg** is used during kernel-mode debugging, the results shown will be for the host computer, and not the target computer. To display raw registry information for the target computer, use the **!reg** extension instead.

Here are some examples. The following will display all subkeys of the specified registry key:

```
dbgcmd  
!dreg hkcu\Software\Microsoft
```

The following will display all values in the specified registry key:

```
dbgcmd  
!dreg System\CurrentControlSet\Services\Tcpip!*
```

The following will display the value Start in the specified registry key:

```
dbgcmd  
!dreg System\CurrentControlSet\Services\Tcpip!Start
```

Typing **!dreg** without any arguments will display some brief Help text for this extension in the Debugger Command window.

Feedback

Was this page helpful?

 Yes

 No

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

!dt

Article • 04/03/2024

The **!dt** extension displays information about a CSR thread.

This extension command should not be confused with the [dt \(Display Type\)](#) command.

```
dbgcmd
```

```
!dt [v] CSR-Thread
```

Parameters

v

Verbose mode.

CSR-Thread

Specifies the hexadecimal address of the CSR thread.

DLL

Ntsdexts.dll

Remarks

This extension displays the thread, process, client ID, flags, and reference count associated with the CSR thread. If verbose mode is selected, the display will also include list pointers, thread handle, and the wait block.

See also

[!dp \(!ntsdexts.dp\)](#)

Feedback

Was this page helpful?

 Yes

 No

!findstack

Article • 04/03/2024

The **!findstack** extension locates all of the stacks that contain a specified symbol or module.

dbgcmd

```
!findstack Symbol [DisplayLevel]
!findstack -?
```

Parameters

Symbol

Specifies a symbol or module.

DisplayLevel

Specifies what the display should contain. This can be any one of the following values. The default value is 1.

0

Displays only the thread ID for each thread that contains *Symbol*.

1

Displays both the thread ID and the frame for each thread that contains *Symbol*.

2

Displays the entire thread stack for each thread that contains *Symbol*.

-?

Displays some brief Help text for this extension in the Debugger Command window.

DLL

Uext.dll

Additional Information

For more information about stack traces, see the [k, kb, kc, kd, kp, kP, kv \(Display Stack Backtrace\) commands](#).

Remarks

The **!stacks** kernel-mode extension also display information about stacks, including a brief summary of the state of every thread.

The following are some examples of the output from this extension:

```
dbgcmd

0:023> !uext.findstack wininet
Thread 009, 2 frame(s) match
    * 06 03eaffac 771d9263 wininet!ICAsyncThread::SelectThread+0x22a
    * 07 03eaffb4 7c80b50b
wininet!ICAsyncThread::SelectThreadWrapper+0xd

Thread 011, 2 frame(s) match
    * 04 03f6ffb0 771cda1d
wininet!AUTO_PROXY_DLLS::DoThreadProcessing+0xa1
    * 05 03f6ffb4 7c80b50b wininet!AutoProxyThreadFunc+0xb

Thread 020, 6 frame(s) match
    * 18 090dfde8 771db73a wininet!CheckForNoNetOverride+0x9c
    * 19 090dfe18 771c5e4d wininet!InternetAutodialIfNotLocalHost+0x220
    * 20 090dfe8c 771c5d6a wininet!ParseUrlForHttp_Fsm+0x135
    * 21 090dfe98 771bcb2c wininet!CFsm_ParseUrlForHttp::RunSM+0x2b
    * 22 090dfeb0 771d734a wininet!CFsm::Run+0x39
    * 23 090dfee0 77f6ad84 wininet!CFsm::RunWorkItem+0x79

Thread 023, 9 frame(s) match
    * 16 0bd4fe00 771bd256 wininet!ICSocket::Connect_Start+0x17e
    * 17 0bd4fe0c 771bcb2c wininet!CFsm_SocketConnect::RunSM+0x42
    * 18 0bd4fe24 771bcada wininet!CFsm::Run+0x39
    * 19 0bd4fe3c 771bd22b wininet!DoFsm+0x25
    * 20 0bd4fe4c 771bd706 wininet!ICSocket::Connect+0x32
    * 21 0bd4fe8c 771bd4cb
wininet!HTTP_REQUEST_HANDLE_OBJECT::OpenConnection_Fsm+0x391
    * 22 0bd4fe98 771bcb2c wininet!CFsm_OpenConnection::RunSM+0x33
    * 23 0bd4feb0 771d734a wininet!CFsm::Run+0x39
    * 24 0bd4fee0 77f6ad84 wininet!CFsm::RunWorkItem+0x79

0:023> !uext.findstack wininet!CFsm::Run 0
Thread 020, 2 frame(s) match
Thread 023, 3 frame(s) match

0:023> !uext.findstack wininet!CFsm 0
Thread 020, 3 frame(s) match
Thread 023, 5 frame(s) match
```

Feedback

Was this page helpful?

 Yes

 No

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

!gatom

Article • 04/03/2024

The **!gatom** extension displays the global atom table.

```
dbgcmd
!gatom
```

DLL

Ntsdexts.dll

Additional Information

For information about the global atom table, see the Microsoft Windows SDK documentation.

Feedback

Was this page helpful?

 Yes

 No

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

!igrep

Article • 10/25/2023

The **!igrep** extension searches for a pattern in disassembly.

dbgcmd

```
!igrep [Pattern [StartAddress]]
```

Parameters

Pattern

Specifies the pattern to search for. If omitted, the last *Pattern* is used.

StartAddress

Specifies the hexadecimal address at which to begin searching. If omitted, the current program counter is used.

DLL

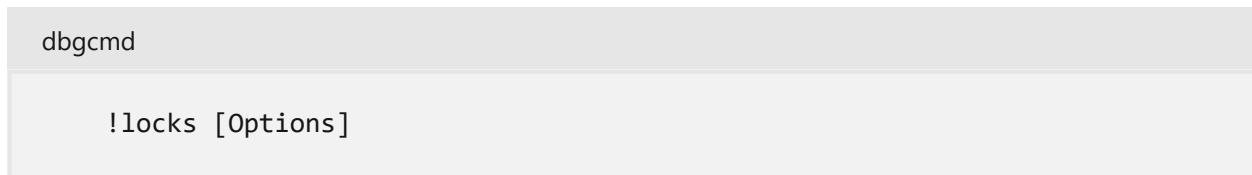
Windows 2000	Ntsdexts.dll
Windows XP and later	Unavailable

!locks (!ntsdexts.locks)

Article • 10/25/2023

The **!locks** extension in Ntsdexts.dll displays a list of critical sections associated with the current process.

This extension command should not be confused with the [**!kdext*.locks**](#) extension command.



Parameters

Options

Specifies the amount of information to be displayed. Any combination of the following options can be used:

-v

Causes the display to include all critical sections, even those that are not currently owned.

-o

Causes the display to only include orphaned information (pointers that do not actually point to valid critical sections).

DLL

Ntsdexts.dll

Additional Information

For other commands and extensions that can display critical section information, see [Displaying a Critical Section](#). For information about critical sections, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension command shows all critical sections that have been initialized by calling `RtlInitializeCriticalSection`. If there are no critical sections, then no output will result.

Here is an example:

```
dbgcmd

0:000> !locks

CritSec w3svc!g_pWamDictator+a0 at 68C2C298
LockCount          0
RecursionCount    1
OwningThread      d1
EntryCount         1
ContentionCount   0
*** Locked

CritSec SMTPSVC+66a30 at 67906A30
LockCount          0
RecursionCount    1
OwningThread      d0
EntryCount         1
ContentionCount   0
*** Locked
```

See also

[Displaying a Critical Section](#)

[CriticalSection Time Outs \(user mode\)](#)

!mapped_file

Article • 04/03/2024

The **!mapped_file** extension displays the name of the file that backs the file mapping that contains a specified address.

```
dbgcmd
```

```
!mapped_file Address
```

Parameters

Address

Specifies the address of the file mapping. If *Address* is not in a mapping, the command fails.

DLL

Uext.dll

The **!mapped_file** extension can only be used during live, nonremote debugging.

Additional Information

For more information about file mapping, see [MapViewOfFile](#) in the Windows SDK.

Remarks

Here are three examples. The first two addresses used are mapped from a file, and the third is not.

```
dbgcmd
```

```
0:000> !mapped_file 4121ec
Mapped file name for 004121ec:
'\Device\HarddiskVolume2\CODE\TimeTest\Debug\TimeTest.exe'
```

```
0:000> !mapped_file 77150000
Mapped file name for 77150000:
'\Device\HarddiskVolume2\Windows\System32\kernel32.dll'
```

```
0:000> !mapped_file 80310000  
No information found for 80310000: error 87
```

Feedback

Was this page helpful?

 Yes

 No

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

!runaway

Article • 04/03/2024

The **!runaway** extension displays information about the time consumed by each thread.

dbgcmd

!runaway [Flags]

Parameters

Flags

Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x1.

Bit 0 (0x1)

Causes the debugger to show the amount of user time consumed by each thread.

Bit 1 (0x2)

Causes the debugger to show the amount of kernel time consumed by each thread.

Bit 2 (0x4)

Causes the debugger to show the amount of time that has elapsed since each thread was created.

DLL

Uext.dll

Ntsdexts.dll

Additional Information

The **!runaway** extension can only be used during live debugging or when debugging crash dump files created by [.dump /mt](#) or [.dump /ma](#).

For information about threads in user mode, see [Controlling Processes and Threads](#). For more information about analyzing processes and threads, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

This extension is a quick way to find out which threads are spinning out of control or consuming too much CPU time.

The display identifies each thread by the debugger's internal thread numbering and by the thread ID in hexadecimal. The debugger IDs are also shown.

Here is an example:

```
dbgcmd

0:001> !runaway 7

User Mode Time
Thread      Time
0:55c      0:00:00.0093
1:1a4      0:00:00.0000

Kernel Mode Time
Thread      Time
0:55c      0:00:00.0140
1:1a4      0:00:00.0000

Elapsed Time
Thread      Time
0:55c      0:00:43.0533
1:1a4      0:00:25.0876
```

Feedback

Was this page helpful?

 Yes

 No

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

!threadtoken

Article • 04/03/2024

The **!threadtoken** extension extension is obsolete.

```
dbgcmd
```

```
!threadtoken
```

DLL

Unavailable

Additional Information

For information about threads and impersonation, see the Microsoft Windows SDK documentation and *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!threadtoken** extension is obsolete in Windows XP and later versions of Windows. Use [!token](#) instead.

If the current thread is impersonating, the token that this thread is using will be displayed.

Otherwise, a message reading "Thread is not impersonating" will appear. The process token will then be displayed.

Tokens will be displayed in the same format that [!handle](#) uses when displaying token handles.

Here is an example:

```
dbgcmd
```

```
0:000> ~
. 0  id: 1d0.55c  Suspend: 1 Teb 7ffde000 Unfrozen
# 1  id: 1d0.1a4  Suspend: 1 Teb 7ffdd000 Unfrozen
```

```
0:000> !threadtoken
```

```
***Thread is not impersonating, using process token***

Auth Id      0 : 0x1c93d
Type         Primary
Imp Level   Anonymous
Token Id    0 : 0x5e8c19
Mod Id      0 : 0x5e8c12
Dyn Chg     0x1f4
Dyn Avail   0x1a4
Groups      26
Privils     17
User        S-1-5-21-2127521184-1604012920-1887927527-74790
Groups      26
S-1-5-21-2127521184-1604012920-1887927527-513
S-1-1-0
S-1-5-32-544
S-1-5-32-545
S-1-5-21-2127521184-1604012920-1887927527-277551
S-1-5-21-2127521184-1604012920-1887927527-211604
S-1-5-21-2127521184-1604012920-1887927527-10546
S-1-5-21-2127521184-1604012920-1887927527-246657
S-1-5-21-2127521184-1604012920-1887927527-277552
S-1-5-21-2127521184-1604012920-1887927527-416040
S-1-5-21-2127521184-1604012920-1887927527-96548
S-1-5-21-2127521184-1604012920-1887927527-262644
S-1-5-21-2127521184-1604012920-1887927527-155802
S-1-5-21-2127521184-1604012920-1887927527-158763
S-1-5-21-2127521184-1604012920-1887927527-279132
S-1-5-21-2127521184-1604012920-1887927527-443952
S-1-5-21-2127521184-1604012920-1887927527-175772
S-1-5-21-2127521184-1604012920-1887927527-388472
S-1-5-21-2127521184-1604012920-1887927527-443950
S-1-5-21-2127521184-1604012920-1887927527-266975
S-1-5-21-2127521184-1604012920-1887927527-158181
S-1-5-21-2127521184-1604012920-1887927527-279435
S-1-5-5-0-116804
S-1-2-0
S-1-5-4
S-1-5-11

Privileges   17
SeUndockPrivilege ( Enabled Default )
SeTakeOwnershipPrivilege ( )
SeShutdownPrivilege ( )
SeDebugPrivilege ( )
SeIncreaseBasePriorityPrivilege ( )
SeAuditPrivilege ( )
SeSyncAgentPrivilege ( )
SeLoadDriverPrivilege ( )
SeSystemEnvironmentPrivilege ( Enabled )
SeRemoteShutdownPrivilege ( )
SeProfileSingleProcessPrivilege ( )
SeCreatePagefilePrivilege ( )
SeCreatePermanentPrivilege ( )
SeSystemProfilePrivilege ( Enabled )
SeBackupPrivilege ( )
```

```
SeMachineAccountPrivilege ( )
SeEnableDelegationPrivilege ( Enabled )
```

Feedback

Was this page helpful?

 Yes

 No

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

!uniqstack

Article • 04/03/2024

The **!uniqstack** extension displays all of the stacks for all of the threads in the current process, excluding stacks that appear to have duplicates.

dbgcmd

```
!uniqstack [ -b | -v | -p ] [ -n ]
```

Parameters

-b

Causes the display to include the first three parameters passed to each function.

-v

Causes the display to include frame pointer omission (FPO) information. On x86-based processors, the calling convention information is also displayed.

-p

Causes the display to include the full parameters for each function that is called in the stack trace. This listing will include each parameter's data type, name, and value. *This requires full symbol information.*

-n

Causes frame numbers to be displayed.

DLL

Uext.dll

Remarks

This extension is similar to the [k, kb, kc, kd, kp, kP, kv \(Display Stack Backtrace\)](#) command, except that it does not display duplicate stacks.

For example:

dbgcmd

```

0:000> !uniqstack
Processing 14 threads, please wait

. 0 Id: f0f0f0f0.15c Suspend: 1 Teb: 00000000`7fff8000 Unfrozen
    Priority: 0
Child-SP          Child-BSP          RetAddr
00000000`0006e5e0 00000000`00070420 00000000`6b009840
00000000`0006e600 00000000`000703d0 00000000`6b03db00
00000000`0006e950 00000000`000703b0 00000000`6b008520
00000000`0006e950 00000000`00070368 00000000`6b1845e0
00000000`0006e9b0 00000000`00070310 00000000`6b009980
00000000`0006e9d0 00000000`000702d0 00000000`6b009ff0
00000000`0006e9e0 00000000`00070248 00000000`77ea4a10
00000000`0006f290 00000000`000700e0 00000000`77ea5d60
00000000`0006f4c0 00000000`00070028 00000000`77ed6000
00000000`0006f550 00000000`00070000 00000000`77ed9000
00000000`0006f550 00000000`00070000 00000000`77ca78a0
00000000`0006ffff0 00000000`00070000 00000000`00000000

. 1 Id: f0f0f0f0.718 Suspend: 1 Teb: 00000000`7fff4000 Unfrozen
    Priority: 0
Child-SP          Child-BSP          RetAddr
00000000`0043eb50 00000000`00440250 00000000`6b008520
00000000`0043eb80 00000000`00440208 00000000`6b1845e0
00000000`0043ebc0 00000000`004401a8 00000000`6b009980
00000000`0043ec00 00000000`00440168 00000000`6b009ff0
00000000`0043ec10 00000000`004400e0 00000000`77ea5f50
00000000`0043f4c0 00000000`00440028 00000000`77ed6000
00000000`0043f550 00000000`00440000 00000000`77ed9000
00000000`0043f550 00000000`00440000 00000000`7de05690
00000000`0043ffff0 00000000`00440000 00000000`00000000

. 13 Id: f0f0f0f0.494 Suspend: 1 Teb: 00000000`7ef98000 Unfrozen
    Priority: 0
Child-SP          Child-BSP          RetAddr
00000000`00feffe0 00000000`00ff0040 00000000`77e94f30
00000000`00feffe0 00000000`00ff0040 00000000`00000000

Total threads: 14
Duplicate callstacks: 11 (windbg thread #s follow):
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

```

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!vadump

Article • 04/03/2024

The **!vadump** extension displays all virtual memory ranges and their corresponding protection information.

```
dbgcmd
```

```
!vadump [-v]
```

Parameters

-v

Causes the display to include information about each original allocation region as well. Because individual addresses within a region can have their protection altered after memory is allocated (by **VirtualProtect**, for example), the original protection status for this larger region may not be the same as that of each range within the region.

DLL

Uext.dll

Additional Information

To view memory protection information for a single virtual address, use [!vprot](#). For information about memory protection, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

Here is an example:

```
dbgcmd
```

```
0:000> !vadump
BaseAddress:      00000000
RegionSize:       00010000
State:            00010000  MEM_FREE
Protect:          00000001  PAGE_NOACCESS
```

```
BaseAddress:      00010000
RegionSize:      00001000
State:           00001000  MEM_COMMIT
Protect:         00000004  PAGE_READWRITE
Type:            00020000  MEM_PRIVATE
.....
```

In this display, the State line shows the state of the memory range beginning at the specified BaseAddress. The possible state values are MEM_COMMIT, MEM_FREE, and MEM_RESERVE.

The Protect line shows the protection status of this memory range. The possible protection values are PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, and PAGE_GUARD.

The Type line shows the memory type. The possible values are MEM_IMAGE, MEM_MAPPED, and MEM_PRIVATE.

Here is an example using the -v parameter:

```
dbgcmd

0:000> !vadump -v
BaseAddress:      00000000
AllocationBase:   00000000
RegionSize:       00010000
State:            00010000  MEM_FREE
Protect:          00000001  PAGE_NOACCESS

BaseAddress:      00010000
AllocationBase:   00010000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:       00001000
State:            00001000  MEM_COMMIT
Protect:          00000004  PAGE_READWRITE
Type:             00020000  MEM_PRIVATE
.....
```

When -v is used, the AllocationProtect line shows the default protection that the entire region was created with. The Protect line shows the actual protection for this specific address.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!vprot

Article • 04/03/2024

The **!vprot** extension displays virtual memory protection information.

```
dbgcmd
```

```
!vprot [Address]
```

Parameters

Address

Specifies the hexadecimal address whose memory protection status is to be displayed.

DLL

Uext.dll

Additional Information

To view memory protection information for all memory ranges owned by the target process, use [!vadump](#). For information about memory protection, see *Microsoft Windows Internals* by Mark Russinovich and David Solomon.

Remarks

The **!vprot** extension command can be used for both live debugging and dump file debugging.

Here is an example:

```
dbgcmd
```

```
0:000> !vprot 30c191c
BaseAddress: 030c1000
AllocationBase: 030c0000
AllocationProtect: 00000080 PAGE_EXECUTE_WRITECOPY
RegionSize: 00011000
State: 00001000 MEM_COMMIT
```

```
Protect: 00000010 PAGE_EXECUTE  
Type: 01000000 MEM_IMAGE
```

In this display, the AllocationProtect line shows the default protection that the entire region was created with. Note that individual addresses within this region can have their protection altered after memory is allocated (for example, if **VirtualProtect** is called). The Protect line shows the actual protection for this specific address. The possible protection values are PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, and PAGE_GUARD.

The State line also applies to the specific virtual address passed to **!vprot**. The possible state values are MEM_COMMIT, MEM_FREE, and MEM_RESERVE.

The Type line shows the memory type. The possible values are MEM_IMAGE, MEM_MAPPED, and MEM_PRIVATE.

Feedback

Was this page helpful?

 Yes

 No

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

Specialized Extensions

Article • 10/25/2023

This section of the reference discusses extension commands in the extension DLLs that are used less often.

In this section

- Storage Kernel Debugger Extensions
- Bluetooth Extensions ([Bthkd.dll](#))
- GPIO Extensions
- USB 3.0 Extensions
- USB 2.0 Debugger Extensions
- RCDRKD Extensions
- HID Extensions
- Logger Extensions ([Logexts.dll](#))
- NDIS Extensions ([Ndiskd.dll](#))
- RPC Extensions ([Rpcexts.dll](#))
- ACPI Extensions ([Acpikd.dll](#) and [Kdexts.dll](#))
- SCSI Miniport Extensions ([Scsikd.dll](#) and [Minipkd.dll](#))
- Windows Driver Framework Extensions ([Wdfkd.dll](#))
- User-Mode Driver Framework Extensions ([Wudfext.dll](#))
- WMI Tracing Extensions ([Wmitrace.dll](#))

Storage Kernel Debugger Extensions

Article • 10/25/2023

The storage kernel debugger extensions (`storagekd`) are used for debugging the storage drivers on Windows 8 and above operating system (OS) targets.

Extension commands that are useful for debugging storage drivers, via `classpnp` managed storage class drivers and `Storport` managed storage miniport drivers, can be found in `Storagekd.dll`.

Please refer to [SCSI Miniport Extensions \(`Scsikd.dll` and `Minipkd.dll`\)](#) for debugging needs for Windows 7 and below version of OS targets.

Important You need special symbols to use this extension. For more information, see [Debugging Tools for Windows](#).

Storage kernel debugger extension commands

Command	Description
<code>!storagekd.storhelp</code>	Displays help text for <code>Storagekd.dll</code> extension commands.
<code>!storagekd.storclass</code>	Displays information about the specified <code>classpnp</code> device.
<code>!storagekd.storadapter</code>	Displays information about the specified <code>Storport</code> adapter.
<code>!storagekd.storunit</code>	Displays information about the specified <code>Storport</code> logical unit.
<code>!storagekd.storloglist</code>	Displays the <code>Storport</code> adapter's internal log entries.
<code>!storagekd.storlogirp</code>	Displays the <code>Storport</code> 's internal log entries for the adapter filtered for the IRP provided.
<code>!storagekd.storlogsrbsrb</code>	Displays the <code>Storport</code> 's internal log entries for the adapter filtered for the Storage (or SCSI) Request Block (SRB) provided.
<code>!storagekd.storsrb</code>	Displays information about the specified Storage (or SCSI) Request Block (SRB).

!storagekd.storadapter

Article • 04/03/2024

The **!storagekd.storadapter** extension displays information about the specified Storport adapter.

```
dbgcmd
```

```
!storagekd.storadapter [Address]
```

Parameters

Address

Specifies the address of a Storport adapter device object. If *Address* is omitted, a list of all Storport adapters is displayed.

DLL

Storagekd.dll

Remarks

Here is an example of the **!storagekd.storadapter** display:

1: kd> !storagekd.storadapter

```
dbgcmd
```

```
# STORPORT adapters:  
=====
```

## Driver	Object	Extension	State
\Driver\vhdmp	fffffa800649a050	fffffa800649a1a0	Working

1: kd> !storagekd.storadapter fffffa800649a050

```
dbgcmd
```

```
ADAPTER  
DeviceObj : fffffa800649a050    AdapterExt: fffffa800649a1a0    DriverObj :  
fffffa800507fcb0  
DeviceState : Working
```

```
LowerDO ffffffa8005f71e10 PhysicalDO ffffffa8005f71e10
SlowLock Free RemLock -666
SystemPowerState: Working AdapterPowerState D0 Full Duplex
Bus 0 Slot 0 DMA 0000000000000000 Interrupt 0000000000000000
Allocated ResourceList 0000000000000000
Translated ResourceList 0000000000000000
Gateway: Outstanding 0 Lower 256 High 256
PortConfigInfo ffffffa800649a2d0
HwInit ffffffa80062e8840 HwDeviceExt ffffffa8004b84d70 (112 bytes)
SrbExt 2256 bytes LUExt 24 bytes
```

Normal Logical Units:

Product	SCSI ID	Object	Extension	Pnd
Out Ct State				

0 0 Msft Virtual Di 0 0 1	fffffa800658a060	fffffa800658a1b0		0
0 0 Working				

Zombie Logical Units:

Product	SCSI ID	Object	Extension	Pnd
Out Ct State				

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storclass

Article • 04/03/2024

The **!storagekd.storclass** extension displays information about the specified *classpnp* device.

dbgcmd

```
!storagekd.storclass [Address [Level]]
```

Parameters

Address

Specifies the address to the device object or device extension of a classpnp device. If *Address* is omitted, a list of all classpnp extensions is displayed.

Level

Specifies the amount of detail to display. This parameter can be set to 0, 1, or 2, with 2 giving the most detail and 0 the least. The default is 0.

DLL

Storagekd.dll

Remarks

Here is an example of the **!storagekd.storclass** display:

1: kd> !storagekd.storclass

dbgcmd

Storage class devices:

```
* !storclass ffffffa80043dc060 [1,2] ST3160812AS Paging Disk
!storclass ffffffa8006581740 [1,2] Msft Virtual Disk Disk
```

Usage: !storclass <class device> <level [0-2]>

1: kd> !storagekd.storclass ffffffa80043dc060 1

```
dbgcmd
```

```
Storage class device ffffffa80043dc060 with extension at ffffffa80043dc1b0
```

```
Classpnp Internal Information at ffffffa8003bec360
```

```
Transfer Packet Engine:
```

Packet	Status	DL Irp	Opcode	Sector/ListId	UL Irp
-----	-----	-----	-----	-----	-----

```
Pending Idle Requests: 0x0
```

```
-- dt classpnp!_CLASS_PRIVATE_FDO_DATA ffffffa8003bec360 --
```

```
Classpnp External Information at ffffffa80043dc1b0
```

```
ST3160812AS 3.ADH 9LS20QRL
```

```
Minidriver information at ffffffa80043dc670
```

```
Attached device object at ffffffa800410a060
```

```
Physical device object at ffffffa800410a060
```

```
Media Geometry:
```

```
Bytes in a Sector = 512
```

```
Sectors per Track = 63
```

```
Tracks / Cylinder = 255
```

```
Media Length = 160000000000 bytes = ~149 GB
```

```
-- dt classpnp!_FUNCTIONAL_DEVICE_EXTENSION ffffffa80043dc1b0 --
```

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storhelp

Article • 04/03/2024

The **!storagekd.storhelp** extension displays help text for Storagekd.dll extension commands.

```
dbgcmd
```

```
!storagekd.storhelp
```

DLL

Storagekd.dll

Remarks

Here is an example of the **!storagekd.storhelp** display:

0: kd> !storagekd.storhelp

```
dbgcmd
```

```
# Storage Debugger Extension
=====
## General Commands
-----
!storhelp      - Displays complete help of the commands provided in this KD
extension
!storclass     - Dumps all class devices managed by classpnp
!storadapter   - Dumps all adapters managed by Storport
!storunit      - Dumps all disks managed by Storport

## STORPORT specific commands
-----
!storlogirp <args>      - displays internal log entries that reference the
specified IRP.
                                See '!storhelp storlogirp' for details.
!storloglist <args>      - displays internal log entries. See '!storhelp
storloglist' for details.
!storlogsrp <args>      - displays internal log entries that reference the
specified SRB.
                                See '!storhelp storlogsrp' for details.
!storsrb <address>      - display details for the specified SCSI or STORAGE
request block
```

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storlogirp

Article • 04/03/2024

The `!storagekd.storlogirp` extension displays the Storport's internal log entries for the adapter filtered for the IRP provided.

dbgcmd

```
!storagekd.storlogirp <Address> <irp> [<starting_entry> [<ending_entry>]] [L  
<count>]
```

Parameters

Address

Specifies the address of a Storport adapter device extension or device object.

irp

The IRP to locate.

starting_entry

The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry

The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count

Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Storagekd.dll

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storloglist

Article • 04/03/2024

The **!storagekd.storloglist** extension displays the Storport adapter's internal log entries.

dbgcmd

```
!storagekd.storloglist <Address> [<starting_entry> [<ending_entry>]] [L  
<count>]
```

Parameters

Address

Specifies the address of a Storport adapter device extension or device object.

starting_entry

The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry

The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count

Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Storagekd.dll

Remarks

Here is an example of **!storagekd.storloglist** display:

0: kd> !storagekd.storloglist fffffe0010f5e01a0

dbgcmd

```
Storport RaidLogList  
Circular buffer location: 0xfffffe0010f5e1720  
Total logs written: 8  
Displaying entries 0 through 7
```

```
-----  
[0]_[23:04:20.521] ResumeDevice..... Caller:  
storport!RaidUnitPauseTimerDpcRoutine+0x28 (fffff800`fb4d0b28), P/P/T/L:  
0/0/0/0, Pause count: 0, Resumed: True  
[1]_[23:04:20.646] ResumeDevice..... Caller:  
storport!RaidUnitPauseTimerDpcRoutine+0x28 (fffff800`fb4d0b28), P/P/T/L:  
0/0/0/0, Pause count: 0, Resumed: True  
[2]_[23:04:20.646] SpPauseDevice..... Caller:  
iaStorAV!RpIPauseDevice+0x67 (fffff800`fb70554f), P/T/L: 3/0/0, Timeout:  
180, Adapter: 0xfffffe0010f5e01a0  
[3]_[23:04:20.646] PauseDevice..... Caller:  
storport!StorPortPauseDevice+0x2f6 (fffff800`fb4b52d6), P/P/T/L: 0/3/0/0,  
Pause count: 1  
[4]_[23:04:20.646] SpResumeDevice..... Caller:  
iaStorAV!RpIResumeDevice+0x5f (fffff800`fb7055fb), P/T/L: 3/0/0, Adapter:  
0xfffffe0010f5e01a0  
[5]_[23:04:20.646] ResumeDevice..... Caller:  
storport!StorPortResumeDevice+0x19 (fffff800`fb4aa23f), P/P/T/L: 0/3/0/0,  
Pause count: 0, Resumed: True  
[6]_[23:04:20.646] SpPauseDevice..... Caller:  
iaStorAV!RpIPauseDevice+0x67 (fffff800`fb70554f), P/T/L: 3/0/0, Timeout:  
180, Adapter: 0xfffffe0010f5e01a0  
[7]_[23:04:20.646] PauseDevice..... Caller:  
storport!StorPortPauseDevice+0x2f6 (fffff800`fb4b52d6), P/P/T/L: 0/3/0/0,  
Pause count: 1
```

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storlogsrb

Article • 04/03/2024

The **!storagekd.storlogsrb** extension displays the Storport's internal log entries for the adapter filtered for the Storage (or SCSI) Request Block (SRB) provided.

dbgcmd

```
!storagekd.storlogsrb <Address> <srb> [<starting_entry> [<ending_entry>]] [L  
<count>]
```

Parameters

Address

Specifies the address of a Storport adapter device extension or device object.

SRB

The SRB to locate.

starting_entry

The beginning entry in the range to display. If not specified, the last *count* entries will be displayed.

ending_entry

The ending entry in the range to display. If not specified, *count* entries will be displayed, beginning with the item specified by *starting_entry*.

count

Count of entries to be displayed. If not specified, a value of 50 is used.

DLL

Storagekd.dll

Feedback

Was this page helpful?

 Yes

 No

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

!storagekd.storsrb

Article • 04/03/2024

The **!storagekd.storsrb** extension displays information about the specified Storage (or SCSI) Request Block (SRB).

```
dbgcmd
```

```
!storagekd.storsrb Address
```

Parameters

Address

Specifies the address of the SRB.

DLL

Storagekd.dll

Remarks

Here is an example of the **!storagekd.storsrb** display:

```
0: kd> !storagekd.storsrb fffffe00111fe25b0
```

```
dbgcmd
```

```
SRB is a STORAGE request block (SRB_EX)
SRB EX 0xfffffe00111fe25b0 Function 28 Version 1, Signature 53524258,
SrbStatus: 0x02[Aborted], SrbFunction 0x00 [EXECUTE SCSI]
Address Type is BTL8

SRB_EX Data Type [SrbExDataTypeScsiCdb16]
[EXECUTE SCSI] SRB_EX: 0xfffffe00111fe2648 OriginalRequest:
0xfffffe001125a9010 DataBuffer/Length: 0xfffffe00112944000 / 0x000000200
PTL: (0, 1, 1) CDB: 28 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00
OpCode: SCSI/READ (10)
```

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!storagekd.storunit

Article • 04/03/2024

The **!storagekd.storunit** extension displays information about the specified Storport logical unit.

```
dbgcmd
```

```
!storagekd.storunit [Address]
```

Parameters

Address

Specifies the address of a Storport unit device object. If *Address* is omitted, a list of all Storport units are displayed.

DLL

Storagekd.dll

Remarks

Here is an example of the **!storagekd.storunit** display:

0: kd> !storagekd.storunit

```
dbgcmd
```

```
# STORPORT Units:  
=====
```

## Product	SCSI ID	Object	Extension	Pnd
Out Ct	State			
Msft	Virtual Di	0 0 1 ffffffa800658a060	ffffffa800658a1b0	0 0
0	Working			

0: kd> !storagekd.storunit ffffffa800658a060

```
dbgcmd
```

```
DO ffffffa800658a060  Ext ffffffa800658a1b0  Adapter ffffffa800649a1a0
Working
  Vendor: Msft          Product: Virtual Disk          SCSI ID: (0, 0, 1)
  Claimed Enumerated
  SlowLock Free   RemLock 1   PageCount 0
  QueueTagList: ffffffa800658a270      Outstanding: Head ffffffa800658a398
  Tail ffffffa800658a398  Timeout -1
  DeviceQueue ffffffa800658a2a0  Depth: 250  Status: Not Frozen
  PauseCount: 0  BusyCount: 0
  IO Gateway: Busy Count 0  Pause Count 0
  Requests: Outstanding 0  Device 0  ByPass 0
```

[Device-Queued Requests]

## IRP	SRB	Type	SRB	XRB	Command
MDL	SGList		Timeout		
-	-	-	-	-	-
-	-	-	-	-	-

[Bypass-Queued Requests]

## IRP	SRB	Type	SRB	XRB	Command
MDL	SGList		Timeout		
-	-	-	-	-	-
-	-	-	-	-	-

[Outstanding Requests]

## IRP	SRB	Type	SRB	XRB	Command
MDL	SGList		Timeout		
-	-	-	-	-	-
-	-	-	-	-	-

[Completed Requests]

IRP	SRB	Type	SRB	XRB	Command
MDL	SGList		Timeout		
-	-	-	-	-	-
-	-	-	-	-	-

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

Bluetooth Extensions (Bthkd.dll)

Article • 10/25/2023

The Bluetooth debugger extensions display information about the current Bluetooth environment on the target system.

Note As you work with the Bluetooth debugging extensions, you may come across undocumented behavior or APIs. We strongly recommend against taking dependencies on undocumented behavior or APIs as it's subject to change in future releases.

In this section

Topic	Description
!bthkd.bthdevinfo	The !bthkd.bthdevinfo command displays the information about a given BTHENUM created device PDO.
!bthkd.bthenuminfo	The !bthkd.bthenuminfo command displays information about the BTHENUM FDO.
!bthkd.bthinfo	The !bthkd.bthinfo command displays details about the BTHPORT FDO. This command is a good starting point for Bluetooth investigations as it displays address information that can be used to access many of the other Bluetooth debug extension commands.
!bthkd.bthhelp	The !bthkd.bthhelp command displays help for the Bluetooth debug extension commands.
!bthkd.bthtree	The !bthkd.bthtree command displays the complete Bluetooth device tree.
!bthkd.bthusbtransfer	The !bthkd.bthusbtransfer command displays the Bluetooth usb transfer context including Irp, Bip and transfer buffer information.
!bthkd.dibflags	The !bthkd.dibflags command displays DEVICE_INFO_BLOCK.DibFlags dumps flags set in _DEVICE_INFO_BLOCK.DibFlags.
!bthkd.hcicmd	The !bthkd.hcicmd command displays a list of the currently pending commands.
!bthkd.hciinterface	The !bthkd.hciinterface command displays the bthport!_HCI_INTERFACE structure.

Topic	Description
!bthkd.l2capinterface	The !bthkd.l2capinterface command displays information about the L2CAP interface.
!bthkd.rfcomminfo	The !bthkd.rfcomminfo command displays information about the RFCOMM FDO and the TDI Device Object.
!bthkd.rfcommconnection	The !bthkd.rfcommconnection command displays information about a given RFCOMM connection object.
!bthkd.rfcommchannel	The !bthkd.rfcommchannel command displays information about a given RFCOMM channel CB.
!bthkd.sdpinterface	The !bthkd.sdpinterface command displays information about the SDP interface.
!bthkd.scointerface	The !bthkd.scointerface command displays information about the SCO interface.
!bthkd.sdpnode	The !bthkd.sdpnode command displays information about a node in an sdp tree.
!bthkd.sdpstream	The !bthkd.sdpstream command displays the contents of a SDP stream.

!bthkd.bthdevinfo

Article • 10/25/2023

The **!bthkd.bthdevinfo** command displays the information about a given BTHENUM created device PDO.

dbgsyntax

```
!bthkd.bthdevinfo addr
```

Parameters

addr

The address of a BTHENUM created device PDO extension.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.bthenuminfo

Article • 10/25/2023

The **!bthkd.bthenuminfo** command displays information about the BTENUM FDO.

dbgsyntax

```
!bthkd.bthenuminfo
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.bthinfo

Article • 10/25/2023

The **!bthkd.bthinfo** command displays details about the BTHPORT FDO. This command is a good starting point for Bluetooth investigations as it displays address information that can be used to access many of the other Bluetooth debug extension commands.

dbgsyntax

```
!bthkd.bthinfo
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.bthhelp

Article • 10/25/2023

The **!bthkd.bthhelp** command displays help for the Bluetooth debug extension commands.

dbgsyntax

```
!bthkd.bthhelp
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.bthtree

Article • 10/25/2023

The **!bthkd.bthtree** command displays the complete Bluetooth device tree.

dbgsyntax

```
!bthkd.bthtree
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.bthusbtransfer

Article • 10/25/2023

The **!bthkd.bthusbtransfer** command displays the Bluetooth usb transfer context including Irp, Bip and transfer buffer information.

dbgsyntax

```
!bthkd.bthusbtransfer addr
```

Parameters

addr

Address of the Bluetooth USB transfer context.

Remarks

You can use the **!bthinfo** command to display the address of USB transfer context. It is listed under the transfer list section.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.dibflags

Article • 10/25/2023

The `!bthkd.dibflags` command displays `DEVICE_INFO_BLOCK.DibFlags` dumps flags set in `_DEVICE_INFO_BLOCK.DibFlags`.

dbgsyntax

```
!bthkd.dibflags flags
```

Parameters

flags

The value of `_DEVICE_INFO_BLOCK.DibFlags` Dumps Flags set in `_DEVICE_INFO_BLOCK.DibFlags`.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.hcicmd

Article • 10/25/2023

The **!bthkd.hcicmd** command displays a list of the currently pending commands.

```
dbgsyntax
```

```
!bthkd.hcicmd
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.hciinterface

Article • 10/25/2023

The **!bthkd.hciinterface** command displays the bthport!_HCI_INTERFACE structure.

```
dbgsyntax
```

```
!bthkd.hciinterface
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.l2capinterface

Article • 10/25/2023

The **!bthkd.l2capinterface** command displays information about the L2CAP interface.

dbgsyntax

```
!bthkd.l2capinterface
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.rfcomminfo

Article • 10/25/2023

The **!bthkd.rfcomminfo** command displays information about the RFCOMM FDO and the TDI Device Object.

dbgsyntax

```
!bthkd.rfcomminfo
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.rfcommconnection

Article • 10/25/2023

The **!bthkd.rfcommconnection** command displays information about a given RFCOMM connection object.

dbgsyntax

```
!bthkd.rfcommconnection addr
```

Parameters

addr

The address of a rfcomm!_RFCOMM_CONN_OBJ structure.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.rfcommchannel

Article • 10/25/2023

The **!bthkd.rfcommchannel** command displays information about a given RFCOMM channel CB.

dbgsyntax

```
!bthkd.rfcommchannel addr
```

Parameters

addr

The address of a rfcomm!_CHANNEL_CB structure.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.sdpinterface

Article • 10/25/2023

The **!bthkd.sdpinterface** command displays information about the SDP interface.

dbgsyntax

```
!bthkd.sdpinterface
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.scointerface

Article • 10/25/2023

The **!bthkd.scointerface** command displays information about the SCO interface.

dbgsyntax

```
!bthkd.scointerface
```

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.sdpnode

Article • 10/25/2023

The **!bthkd.sdpnode** command displays information about a node in an sdp tree.

dbgsyntax

```
!bthkd.sdpnode addr [flags]
```

Parameters

addr

Address of the sdp tree node to display.

flags

0x1 - Recurse node

0x2 - Verbose

default is 0x0

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!bthkd.sdpstream

Article • 10/25/2023

The **!bthkd.sdpstream** command displays the contents of a SDP stream.

dbgsyntax

```
!bthkd.sdpstream streamaddr streamlength
```

Parameters

streamaddr

A pointer to beginning of stream.

streamlength

The length of SDP stream in bytes to display.

DLL

Bthkd.dll

See also

[Bluetooth Extensions \(Bthkd.dll\)](#)

!acxkd

Article • 06/21/2024

The **!acxkd** extension displays information about audio class extension (ACX) drivers. For more information about ACX, see [ACX audio class extensions overview](#).

Syntax

dbgcmd

!acxkd. [Options]

DLL

Acxkd.dll

Debugger Version

The **!acxkd** extension is available in WinDbg version 1.2402.24001.0 and later.

ACX 1.0 debugging

The **!acxkd** debugger extension offers only partial functionality under ACX 1.0. Updating to ACX 1.1 is recommended.

Parameters

Options - Specifies the type of information to be display.

[] [Expand table](#)

Option	Description	Parameter
!help	Displays information on available extension commands.	<code>[<command name>]</code>
!acxcircuit	Dump an ACXCIRCUIT object.	<code><circuit></code> - ACXCIRCUIT WDF handle
!acxdataformat	Dump an ACXDATAFORMAT object.	<code><dataformat></code> - ACXDATAFORMAT

Option	Description	Parameter
		WDF handle
!acxdataformatlist	Dump an ACXDATAFORMATLIST object.	<dataformatlist> - ACXDATAFORMATLIST WDF handle
!acxdevice	Dump an ACXDEVICE object.	<device> - ACXDEVICE WDF handle
!acxelement	Dump an ACXELEMENT object.	<element> - ACXELEMENT WDF handle
!acxevents	Dump events of an ACXObject object.	<events> - ACXObject WDF handle
!acxfactory	Dump an ACXFACTORYCIRCUIT object.	<factory> - ACXFACTORYCIRCUIT WDF handle
!acxmanager	Dump an ACXMANAGER object.	None
!acxmethods	Dump methods of an ACXObject object.	<object> - ACXObject WDF handle
!acxobjbag	Dump an ACXObjectBAG object.	<objbag> - ACXObjectBAG WDF handle
!acxobject	Dump an ACXObject object.	<object> - ACXObject WDF handle
!acxpin	Dump an ACXPIN object.	<pin> - ACXPIN WDF handle
!acxproperties	Dump properties of an ACXObject object.	<properties> - ACXObject WDF handle
!acxstream	Dump an ACXSTREAM object.	<stream> - ACXSTREAM handle
!acxstreambridge	Dump an ACXSTREAMBRIDGE object.	<bridge> - ACXSTREAMBRIDGE handle
!acxtarget	Dump an ACXTARGET object.	<target> - ACXTARGET WDF handle
!acxtemplate	Dump an ACXTEMPLATE object.	<ttmp> - ACXTEMPLATE handle

Remarks

!acxkd.help

To list all available commands, use the acxkd `!help` command.

```
0: kd> !acxkd.help
Commands for C:\Debugger\acxkd.dll:
!acxcircuit          - Dump a ACXCIRCUIT object.
!acxdataformat       - Dump a ACXDATAFORMAT object.
!acxdataformatlist   - Dump a ACXDATAFORMATLIST object.
!acxdevice           - Dump a ACXDEVICE object.
!acxelement          - Dump a ACXELEMENT object.
!acxevents           - Dump events of a ACXObject object.
!acxfactory          - Dump a ACXFACTORYCIRCUT object.
!acxmanager          - Dump a ACXMANAGER object.
!acxmethods          - Dump methods of a ACXObject object.
!acxobjbag           - Dump a ACXObjectBAG object.
!acxobject           - Dump a ACXObject* object.
!acxpin              - Dump a ACXPIN object.
!acxproperties        - Dump properties of a ACXObject object.
!acxstream            - Dump a ACXSTREAM object.
!acxstreambridge     - Dump a ACXSTREAMBRIDGE object.
!acxtarget            - Dump a ACXTARGET* object.
!acxtemplate          - Dump a ACX*TEMPLATE object.
!help                - Displays information on available extension commands
!help <cmd> will give more information for a particular command
```

Use the acxkd `!help` command to learn more about any of the commands, for example the `!acxdevice` command.

```
dbgcmd

0: kd> !acxkd.help acxdevice
!acxdevice <device>
<device> - ACXDEVICE handle
Dump a ACXDEVICE object.
```

Use the `!acxdevice` command as a starting point to examine the ACX driver.

```
dbgcmd

3: kd> !acxdevice 0x00007dfadb0a5358
Dumping info for ACXDEVICE 0x00007dfadb0a5358

In connected standby: FALSE

State: AfxDeviceStateInitialized
State history:
 0 : AfxDeviceStateInvalid
 1 : AfxDeviceStateInvalid
 2 : AfxDeviceStateInvalid
 3 : AfxDeviceStateInvalid
 4 : AfxDeviceStateInvalid
 5 : AfxDeviceStateCreated
 6 : AfxDeviceStateInitializing
```

```

7 : AfxDeviceStateInitialized

Create dispatch list:
    Create name: eHDMIOutTopo
    Dispatch routine: fffff80393179918
    Dispatch context: ffff82052513b960

Circuits:
-----
[Circuit 0]

Name: eHDMIOutTopo
Type: AcxCircuitTypeRender
ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}

State: AfxCircuitStatePoweredUp

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49
48348&0&0002&00000025#{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo

!acxproperties 00007dfad9cccb8
!acxmethode 00007dfad9cccb8
!acxevents 00007dfad9cccb8

# Pins: 2
    !acxpin 00007dfadf996dd8
    !acxpin 00007dfad4697238

# Elements: 1
    !acxelement 00007dfadf997a18

# Streams: 0

!acxcircuit 00007dfad9cccb8

!wdfqueue 00007dfade9beaf8
!wdfdevice 00007dfadb0a5358

!wdfhandle 00007dfadb0a5358
dt Acx01000!Acx::AfxDevice ffff8205256ab420

```

Click on the links in the output to display information using the !acxproperties, !acxmethode and !acxevents commands.

For information on locating the wdfhandle for the ACXDEVICE object, see the [Example ACX driver walkthrough](#) in this article.

WDF commands - !wdfkd.wdfldr

As ACX drivers are WDF drivers, use any of the WDF kernel debugger commands. For example, use `!wdfkd.wdfldr` to display version information and the ACX binding with WDF.

```
dbgcmd

0: kd> !wdfldr acx01000
WDF Driver: Acx01000
-----
CLIENT_MODULE 0xfffff82052abdc0
    WDF Version v1.31
    ImageName   Acx01000.sys
    ImageAddress 0xfffff80393150000
    ImageSize    0xb3000
    BindingList  0xfffff82052abdcc08

    ImageName      WdfVer Ver   WdfGlobals           BindInfo
    ImageAddress   ImageSize
    Wdf01000.sys   v1.33  v1.33  0xfffff8205218d8fb0 0xfffff8205218d8df0
0xfffff80356a00000 0x000c7000
-----
CLASS_MODULE 0xfffff820525b3dc90
    WDF Version v1.31
    Version     v1.1
    Service     \Registry\Machine\System\CurrentControlSet\Services\acx01000
    ImageName   Acx01000.sys
    ImageAddress 0xfffff80393150000
    ImageSize    0xb3000
    ClientsList  0xfffff820525b3dcf8
    Associated Clients: 1

    ImageName      WdfVer Ver   WdfGlobals           BindInfo
    ImageAddress   ImageSize
    AcxHdAudio.sys v1.25  v1.0   0xfffff820527f70ae0 0xfffff803930df3e8
0xfffff803930c0000 0x0008a000
-----
```

!acxkd.acxmanager

Use the `!acxmanager` command to display information about the ACXMANAGER object. This provides a good starting point to investigate ACX drivers.

This example shows the first part of the extensive `!acxmanager` output provided for a multircuit ACX configuration.

```
dbgcmd

10: kd> !acxmanager
Dumping info for ACXMANAGER 0x000054f94d1d4378
```

```
Delete pending: No

# singleton composites: 8
-----
[Composite 0]

State: AfxCompositeStateActive
!acxobjbag 000054f94c8e61c8
!acxtemplate 000054f94c014d28
!acxobject 000054f94c0141c8

# circuits: 3
-----
[Circuit 0 CORE]
...
```

This example output shows a single ACX circuit.

```
dbgcmd

0: kd> !acxmanager
Dumping info for ACXMANAGER 0x000049f6c3c769f8

Delete pending: No

# singleton composites: 0
# Composite factories: 0
!wdfhandle 000049f6c3c769f8
dt Acx01000!Acx::AfxManager fffffb6093c3896b0
```

!acxkd.acxobject

In the output for the !acxmanager an address is provided for the wdfhandle. Use the wdfhandle address with the `!acxobject` command to display information about the ACXMANAGER or any other ACX object.

```
dbgcmd

0: kd> !acxobject 000049f6c3c769f8
Dumping info for ACXMANAGER 0x000049f6c3c769f8

Delete pending: No

# singleton composites: 0
# Composite factories: 0
!wdfhandle 000049f6c3c769f8
dt Acx01000!Acx::AfxManager fffffb6093c3896b0
```

Click on the `dt` link in the output shown above, to see more information about the internal ACX object structures.

```
0: kd> dt Acx01000!Acx::AfxManager fffffb6093c3896b0
+0x000 m_Object          : 0x000049f6`c3c769f8 ACXMANAGER_
=fffffb803`8a478ad8 s_AfxManager      : 0xfffffb609`3c3896b0 Acx::AfxManager
+0x008 m_AcxGlobals       : 0xfffffb609`403cae24 _ACX_DRIVER_GLOBALS
+0x010 m_Flags           : 0
+0x010 m_Unloading       : 0y0
+0x018 m_CompositesMutex : _FAST_MUTEX
+0x050 m_CompositeFactoriesList : _LIST_ENTRY [ 0xfffffb609`3c389700 -
0xfffffb609`3c389700 ]
+0x060 m_CompositeFactoriesCount : 0n0
+0x068 m_CompositesList  : _LIST_ENTRY [ 0xfffffb609`3c389718 -
0xfffffb609`3c389718 ]
+0x078 m_CompositesCount : 0n0
+0x080 m_FactoriesList   : _LIST_ENTRY [ 0xfffffb609`3c389730 -
0xfffffb609`3c389730 ]
+0x090 m_CircuitsList    : _LIST_ENTRY [ 0xfffffb609`40a27068 -
0xfffffb609`40a27068 ]
+0x0a0 m_FiltersMutex    : _FAST_MUTEX
+0x0d8 m_FactoryFiltersList : _LIST_ENTRY [ 0xfffffb609`3c389788 -
0xfffffb609`3c389788 ]
+0x0e8 m_CircuitFiltersList : _LIST_ENTRY [ 0xfffffb609`3c389798 -
0xfffffb609`3c389798 ]
+0x0f8 m_FactoriesNotificationHandle : 0xfffffc70c`b2356790 Void
+0x100 m_CircuitsNotificationHandle : 0xfffffc70c`b2357ec0 Void
+0x108 m_CommandQueue     : 0xfffffb609`3ffc5400 Acx::AfxWorkQueue
```

Note that the debugger output may contain a mix of public ACX objects such as ACXMANAGER, ACXCIRCUITFACTORY, and ACXCIRCUIT and internal structures that are defined to be opaque. The internal types are not guaranteed to stay the same, or be available in different releases of ACX, and must not be called or used directly.

Since ACX objects are WDF objects, you can use the `!wdfkd.wdfhandle` command to display additional information about the ACXMANAGER object.

```
0: kd> !wdfhandle 000049f6c3c769f8
Treating handle as a KMDF handle!

Dumping WDFHANDLE 0x000049f6c3c769f8
=====
Handle type is WDOBJECT [ACXMANAGER]
RefCount: 2
Contexts:
    context: dt 0xfffffb6093c3896b0 Acx01000!AfxManager (size is 0x110)
```

```
bytes)
    EvtCleanupCallback ffffff8038a453070
Acx01000!Acx::WdfCpp::ObjectContext<ACXMANAGER__>
* ,Acx::AfxManager>::EvtObjectContextCleanupThunk
    EvtDestroyCallback ffffff8038a453000
Acx01000!Acx::WdfCpp::ObjectContext<ACXMANAGER__>
* ,Acx::AfxManager>::EvtObjectContextDestroyThunk

    context: dt 0xfffffb609404c2280 Acx01000!WdfCustomType_ACXMANAGER (size
is 0x10 bytes)
    <no associated attribute callbacks>

Parent: !wdfhandle 0x000049f6d4e2d5c8, type is WDFDEVICE
Owning device: !wdfdevice 0x000049f6d4e2d5c8

!wdfobject 0xfffffb6093c389600
```

!acxkd.acxpin

Commands that display other ACX information, such as `!acxpin` require a WDF handle to the object. For information on locating the WDF handle for an ACX object, see [Example ACX driver walkthrough](#) in this article.

```
dbgcmd

0: kd> !acxpin 0x000049f6befeee38
Dumping info for ACXPIN 0x000049f6befeee38

ID: 0
Type: AcxPinTypeSink
Type: AcxPinCommunicationSink

Category: KSCATEGORY_AUDIO
Name: {00000000-0000-0000-0000-000000000000}
```

Depending on the state of the ACX object, not all information may be available to display.

!acxkd.acxdataformatlist

Similar to `!acxpin`, `!acxdataformatlist` displays information on ACX dataformat lists.

```
dbgcmd

0: kd> !acxdataformatlist 0x000049f6bf8be668
Dumping info for ACXDATAFORMATLIST 0x000049f6bf8be668

# Scan count: 0
```

```
# Data formats: 6
Data formats:
    Sample Rate: 48000, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3 (default)
        Sample Rate: 48000, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3
        Sample Rate: 44100, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3
        Sample Rate: 44100, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3
        Sample Rate: 32000, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3
        Sample Rate: 32000, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3

!wdfhandle 000049f6bf8be668
dt Acx01000!Acx::AfxDataFormatList fffffb60940444530
```

Example ACX driver walkthrough

This section provides a walkthrough of debugging an ACX driver.

Symbols path

Use [.symfix](#) and the [.sympath \(Set Symbol Path\)](#) commands to change the symbol path. If you're using local code with the driver add the path to that code as well. Use the [.reload \(Reload Module\)](#) command to reload symbols from the current path.

```
dbgcmd

.symfix
.sympath+ C:\Windows-driver-samples-
develop\audio\Acx\Samples\AudioCodec\Driver
.reload /f
```

Debugger context for drivers

If you're debugging an active ACX driver, set a breakpoint. This places the debugger in the context of the ACX objects, to be able to gather and display information.

These example breakpoints are designed to fire as the sample AudioCodec driver starts up.

```
dbgcmd
```

```
bm AudioCodec!DriverEntry  
bm AudioCodec!AcxDriverInitialize
```

These example breakpoints are designed to fire when specific actions occur, such as pin or circuit creation.

```
dbgcmd  
  
bm AudioCodec!AcxPinCreate  
bm AudioCodec!AcxCircuitCreate  
bm AudioCodec!Codec_EvtBusDeviceAdd
```

Once the driver is loaded and the appropriate breakpoint has fired, and valid execution context is available, use !acxkd commands to display information about any ACX object. Use the !acxobject command for general information and a specific command, such as !acxcircuit or !acxpin for more granular information.

Load the acxkd dll

Use the [.load \(Load Extension DLL\)](#) command to load the acxkd.dll extension.

```
dbgcmd  
  
.load acxkd.dll
```

Display information about the ACX driver

To gather information about the target driver, use the [!m \(List Loaded Modules\)](#) command to see all loaded drivers. Then use the Dvm options to display information about the ACX driver of interest as shown here.

```
dbgcmd  
  
0: kd> !m Dvm AcxHdAudio  
Browse full module list  
start end module name  
fffff803`8a3c0000 fffff803`8a448000 AcxHdAudio (private pdb symbols)  
C:\ProgramData\Debug\sym\AcxHdAudio.pdb\6AEA2622909B20C1AD149C57ACBB4A6F1\AcxH  
dAudio.pdb  
    Loaded symbol image file: AcxHdAudio.sys  
    Mapped memory image file:  
C:\ProgramData\Debug\sym\AcxHdAudio.sys\082942338800\AcxHdAudio.sys  
        Image path: \SystemRoot\System32\drivers\AcxHdAudio.sys  
        Image name: AcxHdAudio.sys
```

```
Browse all global symbols  functions  data  Symbol Reload
Image was built with /Bprepro flag.
Timestamp:          08294233 (This is a reproducible build file hash, not
a timestamp)
CheckSum:           00087DD6
ImageSize:          00088000
Translations:       0000.04b0 0000.04e4 0409.04b0 0409.04e4
Information from resource tables:
```

Use the [x \(Examine Symbols\)](#) command and a wildcard mask to display specific ACX structures, such as an ACXPIN.

```
dbgcmd
```

```
0: kd> x /D AcxHdAudio!acxpin*
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

fffff803`8a3e3216 AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+66
fffff803`8a3e31e0 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+30
fffff803`8a3e361f AcxHdAudio!AcxPinNotifyDataFormatChange = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+46f
fffff803`8a3e7396 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+962
fffff803`8a3e74a2 AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+a6e
fffff803`8a3e75f8 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+bc4
fffff803`8a3e789a AcxHdAudio!AcxPinAddJacks = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+e66
fffff803`8a3e5ae5 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodecR_EvtAcxPinRetrieveJackSinkInfo+25
fffff803`8a3e5913 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodecR_EvtAcxPinRetrieveName+73
fffff803`8a3ea277 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+b27
fffff803`8a3ea39c AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+c4c
fffff803`8a3ea4bc AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+d6c
fffff803`8a3ea7b2 AcxHdAudio!AcxPinAddJacks = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+1062
```

Depending on the current execution context in the debugger, it may be possible to use the [dx \(Display Debugger Object Model Expression\)](#) to drill down into specific ACX structures.

```
dbgcmd
```

```
0: kd> dx -r2 AudioCodec!AcxDeviceAddCircuit
AudioCodec!AcxDeviceAddCircuit          :
AudioCodec!AcxDeviceAddCircuit+0x0 [Type: long __cdecl(WDFDEVICE_
*,ACXCIRCUIT_ *)]
0: kd> u ffffff8007ead1120
AudioCodec!AcxDeviceAddCircuit [C:\Program Files (x86)\Windows
Kits\10\Include\10.0.26016.0\km\acx\km\1.1\AcxDevice.h @ 206]:
fffff800`7ead1120 4889542410      mov     qword ptr [rsp+10h],rdx
fffff800`7ead1125 48894c2408      mov     qword ptr [rsp+8],rcx
fffff800`7ead112a 4883ec38       sub    rsp,38h
fffff800`7ead112e b808000000      mov     eax,8
fffff800`7ead1133 486bc046       imul   rax,rax,46h
fffff800`7ead1137 488d0dc2ab0000  lea    rcx,[AudioCodec!AcxFunctions
(fffff800`7eadbd00)]
fffff800`7ead113e 488b0401       mov     rax,qword ptr [rcx+rax]
fffff800`7ead1142 4889442420      mov     qword ptr [rsp+20h],rax
```

!wdfkd.wdfdriverinfo

Use the [!wdfdriverinfo](#) command with the name of the driver to gather WDF information, such as the associated ACX objects and ACXDEVICE wdfhandle.

```
dbgcmd

0: kd> !wdfdriverinfo AcxHdAudio.sys 1 -v
-----
Default driver image name: AcxHdAudio
WDF library image name: Wdf01000
  FxDriverGlobals 0xfffffb609403e3de0
  WdfBindInfo      0xfffff8038a3dd1b0
    Version        v1.25
  Library module  0xfffffb60929cc6050
    ServiceName
\Registry\Machine\System\CurrentControlSet\Services\Wdf01000
  ImageName       Wdf01000
-----
WDFDRIVER: 0x000049f6bfe9d5d8
  context: dt 0xfffffb60940162bc0 AcxHdAudio!CODEC_DRIVER_CONTEXT (size is
0x1 bytes)
    <no associated attribute callbacks>

  context: dt 0xfffffb6093ccead20 Acx01000!AfxDriver (size is 0x20 bytes)
    EvtCleanupCallback ffffff8038a455780
Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_>
*,Acx::AfxDriver>::EvtObjectContextCleanupThunk
    EvtDestroyCallback ffffff8038a455740
Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_>
*,Acx::AfxDriver>::EvtObjectContextDestroyThunk
Object Hierarchy: !wdfhandle 0x000049f6bfe9d5d8 0xff
Driver logs: !wdflogdump AcxHdAudio.sys -d
Framework logs: !wdflogdump AcxHdAudio.sys -f
```

```

!wdfdevice 0x000049f6bffba488 ff (FDO)
    Pnp/Power State: WdfDevStatePnpStarted, WdfDevStatePowerD0,
    WdfDevStatePwrPolStartedWakeCapable
        context: dt 0xfffffb60940045e60 AcxHdAudio!CODEC_DEVICE_CONTEXT
    (size is 0x110 bytes)
        EvtCleanupCallback fffff8038a3e3c90
    AcxHdAudio!HDACodec_EvtDeviceContextCleanup

        context: dt 0xfffffb60933f030f0 Acx01000!AfxDevice (size is 0x150
    bytes)
        EvtCleanupCallback fffff8038a451910
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextCleanupThunk
        EvtDestroyCallback fffff8038a451740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextDestroyThunk

        context: dt 0xfffffb60940a26b90
    AcxHdAudio!CODEC_RENDER_DEVICE_CONTEXT (size is 0x38 bytes)
        <no associated attribute callbacks>

        context: dt 0xfffffb609409f2e00
    AcxHdAudio!CODEC_CAPTURE_DEVICE_CONTEXT (size is 0x8 bytes)
        <no associated attribute callbacks>
    !wdfdevicequeues 0x000049f6bffba488

    !wdfdevice 0x000049f6bef1a848 ff (PDO)
        Pnp/Power State: WdfDevStatePnpStarted,
    WdfDevStatePowerD0BusWakeOwner, WdfDevStatePwrPolStartedWakeCapable
        context: dt 0xfffffb609410e5aa0
    AcxHdAudio!CODEC_RENDER_DEVICE_CONTEXT (size is 0x38 bytes)
        EvtCleanupCallback fffff8038a3e6400
    AcxHdAudio!HDACodecR_EvtDeviceContextCleanup

        context: dt 0xfffffb60933f090d0 Acx01000!AfxDevice (size is
    0x150 bytes)
        EvtCleanupCallback fffff8038a451910
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextCleanupThunk
        EvtDestroyCallback fffff8038a451740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextDestroyThunk
    !wdfdevicequeues 0x000049f6bef1a848

-----
WDF Verifier settings for AcxHdAudio.sys is OFF
-----

```

In the output shown above, a link is available to the associated wdfdevice. Click on that link to display information about the associated WDF device objects.

```
dbgcmd
```

```
0: kd> !wdfdevice 0x000049f6bef1a848 ff
Treating handle as a KMDF handle!

Dumping WDFDEVICE 0x000049f6bef1a848
=====
WDM PDEVICE_OBJECTs: self fffffb60940ddbdd0

Pnp state: 119 ( WdfDevStatePnpStarted )
Power state: 309 ( WdfDevStatePowerD0BusWakeOwner )
Power Pol state: 531 ( WdfDevStatePwrPolStartedWakeCapable )
```

!acxkd.acxdevice

Using the same wdfhandle with `!acxdevice` provides ACX centric information.

```
dbgcmd
```

```
3: kd> !acxdevice 0x00007dfadb0a5358
Dumping info for ACXDEVICE 0x00007dfadb0a5358

In connected standby: FALSE

State: AfxDeviceStateInitialized
State history:
 0 : AfxDeviceStateInvalid
 1 : AfxDeviceStateInvalid
 2 : AfxDeviceStateInvalid
 3 : AfxDeviceStateInvalid
 4 : AfxDeviceStateInvalid
 5 : AfxDeviceStateCreated
 6 : AfxDeviceStateInitializing
 7 : AfxDeviceStateInitialized

Create dispatch list:
  Create name: eHDMIOutTopo
  Dispatch routine: fffff80393179918
  Dispatch context: ffff82052513b960

Circuits:
-----
[Circuit 0]

  Name: eHDMIOutTopo
  Type: AcxCircuitTypeRender
  ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}

  State: AfxCircuitStatePoweredUp
```

```

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49
48348&0&0002&00000025#\{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo

!acxproperties 00007dfad9ccccb8
!acxmethods 00007dfad9ccccb8
!acxevents 00007dfad9ccccb8

# Pins: 2
    !acxpin 00007dfadf996dd8
    !acxpin 00007dfad4697238

# Elements: 1
    !acxelement 00007dfadf997a18

# Streams: 0

!acxcircuit 00007dfad9ccccb8

!wdfqueue 00007dfade9beaf8
!wdfdevice 00007dfadb0a5358

!wdfhandle 00007dfadb0a5358
dt Acx01000!Acx::AfxDevice ffff8205256ab420

```

To display information about the other ACX objects, such as the ACXCIRCUIT, use the link in output above that invokes `!acxcircuit` with the appropriate wdfhandle.

dbgcmd

```

3: kd> !acxcircuit 00007dfad9ccccb8
Dumping info for ACXCIRCUIT 0x00007dfad9ccccb8

Name: eHDMIOutTopo
Type: AcxCircuitTypeRender
ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}

State: AfxCircuitStatePoweredUp
State history:
    0 : AfxCircuitStateInvalid
    1 : AfxCircuitStateInvalid
    2 : AfxCircuitStateInvalid
    3 : AfxCircuitStateCreated
    4 : AfxCircuitStateInitializing
    5 : AfxCircuitStateInitialized
    6 : AfxCircuitStatePoweredDown
    7 : AfxCircuitStatePoweredUp

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49

```

```
48348&0&0002&00000025#{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo
```

```
# Power references: 0
# Open handles: 18

!acxproperties 00007dfad9ccccb8
!acxmethods 00007dfad9ccccb8
!acxevents 00007dfad9ccccb8

# Pins: 2
!acxpin 00007dfadf996dd8
!acxpin 00007dfad4697238

# Elements: 1
!acxelement 00007dfadf997a18
```

!acxkd.acxproperties

In the output command links with the wdfhandle are provided for other objects such as ACX properties, that can be displayed.

```
dbgcmd
```

```
0: kd> !acxproperties 000049f6bf436b88
Dumping properties info for ACXObject 0x000049f6bf436b88

# sets: 4

Set: {8C134960-51AD-11CF-878A-94F801C10000}
    Id: 0, Flags: 0x1
    Id: 1, Flags: 0x1

Set: {720D4AC0-7533-11D0-A5D6-28DB04C10000}
    Id: 0, Flags: 0x1
    Id: 1, Flags: 0x1
    Id: 2, Flags: 0x1
    Id: 3, Flags: 0x1

Set: {C034FDB0-FF75-47C8-AA3C-EE46716B50C6}
    Id: 1, Flags: 0x1
    Id: 2, Flags: 0x1
    Id: 3, Flags: 0x1

Set: {4D12807E-55DB-48B8-A466-F15A510F5817}
    Id: 1, Flags: 0x1
```

!wdfhandle

Also available in the `!wdfdriverinfo` output is a link to a wdfhandle for the WDF object hierarchy associated with ACX.

```
dbgcmd
```

```
Object Hierarchy: !wdfhandle 0x000049f6bfe9d5d8 0xff
```

Clicking on that link displays the WDF object hierarchy for the ACX driver. This output can be used to locate WDF handles for other ACX and WDF objects.

```
dbgcmd
```

```
0: kd> !wdfhandle 0x000049f6bfe9d5d8 0xff
Treating handle as a KMDF handle!

Dumping WDFHANDLE 0x000049f6bfe9d5d8
=====
Handle type is WDFDRIVER
RefCount: 1
Contexts:
    context: dt 0xfffffb60940162bc0 AcxHdAudio!CODEC_DRIVER_CONTEXT (size is
0x1 bytes)
        <no associated attribute callbacks>

    context: dt 0xfffffb6093ccead20 Acx01000!AfxDriver (size is 0x20 bytes)
        EvtCleanupCallback ffffff8038a455780
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_*
*,Acx::AfxDriver>::EvtObjectContextCleanupThunk
        EvtDestroyCallback ffffff8038a455740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_*
*,Acx::AfxDriver>::EvtObjectContextDestroyThunk

Child WDFHANDLES of 0x000049f6bfe9d5d8:
    !wdfhandle 0x000049f6bfe9d5d8  dt FxDriver 0xfffffb60940162a20 Context
fffffb60940162bc0, Context fffffb6093ccead20 Cleanup ffffff8038a455780 Destroy
ffffff8038a455740
        !wdfdevice 0x000049f6bffba488  dt FxDevice 0xfffffb60940045b70
    Context fffffb60940045e60 Cleanup ffffff8038a3e3c90, Context fffffb60933f030f0
    Cleanup ffffff8038a451910 Destroy ffffff8038a451740, Context fffffb60940a26b90,
    Context fffffb609409f2e00
        WDF INTERNAL      dt FxDefaultIrpHandler 0xfffffb6093fa54130
        WDF INTERNAL      dt FxPkgGeneral 0xfffffb609401f5610
        WDF INTERNAL      dt FxWmiIrpHandler 0xfffffb609401f6510
        WDF INTERNAL      dt FxPkgIo 0xfffffb6093f6d2400
            !wdfqueue 0x000049f6c00114b8  dt FxIoQueue
0xfffffb6093ffeeb40
        WDF INTERNAL      dt FxPkgFdo 0xfffffb6093f2ae020
            !wdfhandle 0x000049f6bfe099f8  dt FxCmResList 0xfffffb609401f6600
            !wdfhandle 0x000049f6bfe0a358  dt FxCmResList 0xfffffb609401f5ca0
            !wdfchildlist 0x000049f6c09adb98  dt FxChildList
0xfffffb6093f652460
            !wdfiotarget 0x000049f6c092d1d8  dt FxIoTarget
```

```

0xfffffb6093f6d2e20
    !wdfqueue 0x000049f6c04904c8  dt FxIoQueue 0xfffffb6093fb6fb30
Context fffffb6093fa8dc0 Cleanup fffff8038a45a350 Destroy fffff8038a45a310
        !wdfhandle 0x000049f6c09e7ed8  dt FxWorkItem 0xfffffb6093f618120
Context fffffb6093f618220
    WDF INTERNAL  dt FxWmiProvider 0xfffffb609403d9b50
        WDF INTERNAL  dt FxWmiInstanceExternal 0xfffffb60940a11320
        WDF INTERNAL  dt FxWmiProvider 0xfffffb609403d9300
        WDF INTERNAL  dt FxWmiInstanceExternal 0xfffffb60940a11720
    !wdfdevice 0x000049f6bef1a848  dt FxDevice 0xfffffb609410e57b0
Context fffffb609410e5aa0 Cleanup fffff8038a3e6400, Context fffffb60933f090d0
Cleanup fffff8038a451910 Destroy fffff8038a451740
    WDF INTERNAL  dt FxDefaultIrpHandler 0xfffffb60940c16450
    WDF INTERNAL  dt FxPkgGeneral 0xfffffb60940bc9a10
    WDF INTERNAL  dt FxWmiIrpHandler 0xfffffb60940bc9380
    WDF INTERNAL  dt FxPkgIo 0xfffffb609400ab5c0
        !wdfqueue 0x000049f6beef588  dt FxIoQueue
0xfffffb60941102a70
    WDF INTERNAL  dt FxPkgPdo 0xfffffb609410e6020
    !wdfhandle 0x000049f6bf436e58  dt FxCmResList 0xfffffb60940bc91a0
    !wdfhandle 0x000049f6bf436d68  dt FxCmResList 0xfffffb60940bc9290
    !wdfqueue 0x000049f6bef185c8  dt FxIoQueue 0xfffffb609410e7a30
Context fffffb60940e72ed0 Cleanup fffff8038a45a350 Destroy fffff8038a45a310
        !wdfhandle 0x000049f6bf436b88 [ACXCIRCUIT]  dt FxUserObject
0xfffffb60940bc9470 Context fffffb60940bc9520, Context
...

```

WDF log commands

[!wdflogdump](#) can be useful for troubleshooting an ACX driver by displaying WDF log information.

Display the log for ACX using the -d driver option.

```

dbgcmd

0: kd> !wdflogdump acx01000 -d
Log dump command
=====
!wdflogdump Acx01000 -a 0xFFFF820527861000 AFX_Client1      4096
!wdflogdump Acx01000 -a 0xFFFF8205215C0000 AFX_Log          4096
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\Dbg\sym\Acx01000.pdb\B13D39B43205B60C07935803D7CB96981\Acx010
00.pdb
--- start of log ---
AFX_Client1 1: Acx::AfxCircuit::Register - INFO:ACXCIRCUIT 00007DFAD9CCCCB8
Registered
AFX_Client1 2: Acx::AfxCircuit::PowerUpNotification - INFO:ACXCIRCUIT

```

```
00007DFAD9CCCCB8, EvtAcxCircuitPowerUp callback, STATUS_SUCCESS
AFX_Client1 5: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 7: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 8: Acx::AfxMute::EvtMuteEventEnableCallback - INFO:ACXMUTE
00007DFADF997A18, enabled ACXEVENT 00007DFADF996F98
AFX_Client1 10: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 12: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 13: Acx::AfxPin::EvtJackEventEnableCallback - INFO:ACXPIN
00007DFAD4697238, enabled ACXEVENT 00007DFADF9973F8
AFX_Client1 15: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 17: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
```

Use [!wdflogdump](#) to display the framework log for a specific ACX driver using the **-f** option.

```
dbgcmd

0: kd> !wdflogdump AcxHdAudio -f
Trace searchpath is:

Trace format prefix is: %7!u!: %4!s! %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\Txt\sym\Wdf01000.pdb\CBDEA3A4F64C17C1752E652A91DD14761\Wdf010
00.pdb
Gather log: Please wait, this may take a moment (reading 4024 bytes).
% read so far ... 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
There are 65 log entries
--- start of log ---
16131: 09/03/2023-23:43:07.3233594 imp_WdfRegistryOpenKey - new WDFKEY
object open failed, 0xc0000034(STATUS_OBJECT_NAME_NOT_FOUND)
16132: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleDecrementIo from FxIdleBusy
16133: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleStartTimer from FxIdleDecrementIo
16134: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleTimerRunning from FxIdleStartTimer
16135: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
```

```
state FxIdleCancelTimer from FxIdleTimerRunning
16136: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleCheckIoCount from FxIdleCancelTimer
16137:
...
...
```

See also

For more information, see [Kernel Streaming Debugging](#). For a walkthrough of debugging with a WDM audio driver, see [Debug Drivers - Step by Step Lab \(Sysvad Kernel Mode\)](#). For more information about ACX, see [ACX audio class extensions overview](#).

Feedback

Was this page helpful?



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

GPIO Extensions

Article • 10/25/2023

The General Purpose Input/Output (GPIO) extension commands display the software state of GPIO controllers. These commands display information from data structures maintained by the GPIO framework extension driver (Msgpioclx.sys). For information about the GPIO framework extension, see [General-Purpose I/O \(GPIO\) Drivers](#).

The GPIO debugger extension commands are implemented in gpiokd.dll. To load the GPIO commands, enter `.load gpiokd.dll` in the debugger.

Each GPIO controller has a set of banks. Each bank has a pin table that has an array of pins. The GPIO debugger extension commands display information about GPIO controllers, banks, pin tables, and pins.

Data structures used by the GPIO commands

The GPIO debugger extension commands use these data structures, which are defined by Msgpioclx.sys.

`msgpioclx!_DEVICE_EXTENSION`

The device extension structure for the GPIO framework extension driver. This structure holds information about an individual GPIO controller.

`msgpioclx!_GPIO_BANK_ENTRY`

This structure holds information about an individual bank of a GPIO controller.

`msgpioclx!_GPIO_PIN_INFORMATION_ENTRY`

This structure holds information about an individual pin in a bank of a GPIO controller.

Getting started with GPIO debugging

To start debugging a GPIO issue, enter the [`!gpiokd.clientlist`](#) command. The `!gpiokd.clientlist` command displays an overview of all registered GPIO controllers and displays addresses that you can pass to other GPIO debugger commands.

In this section

Topic	Description
!gpiokd.help	The !gpiokd.help command displays help for the GPIO debugger extension commands.
!gpiokd.bankinfo	The !gpiokd.bankinfo command displays information about a GPIO bank.
!gpiokd.clientlist	The !gpiokd.clientlist command displays all registered GPIO controllers.
!gpiokd.gpioext	The !gpiokd.gpioext command displays information about a GPIO controller.
!gpiokd.pininfo	The !gpiokd.pininfo command displays information about a specified GPIO pin.
!gpiokd.pinisrvec	The !gpiokd.pinisrvec command displays Interrupt Service Routine (ISR) vector information for a specified pin.
!gpiokd.printable	The !gpiokd.printable command displays information about an array of GPIO pins.

See also

[Specialized Extension Commands](#)

!gpiokd.help

Article • 04/03/2024

The **!gpiokd.help** extension displays help for the [GPIO debugger extension commands](#).

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!gpiokd.bankinfo

Article • 04/03/2024

The **!gpiokd.bankinfo** extension displays information about a GPIO bank.

dbgcmd

```
!gpiokd.bankinfo BankAddress [Flags]
```

Parameters

BankAddress

Address of a [_GPIO_BANK_ENTRY](#) structure that represents a bank of a GPIO controller.

Flags

Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

[+] Expand table

Flag	Description
0x1	Displays the pin table.
0x2	Displays the Enable and Mask registers.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!gpiokd.clientlist

Article • 04/03/2024

The **!gpiokd.clientlist** extension displays all registered GPIO controllers.

dbgcmd

!gpiokd.clientlist [Flags]

Parameters

Flags

Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

[] [Expand table](#)

Flag	Description
0x1	For each controller, displays detailed information including all of its banks.
0x2	If bit 0 (0x1) is set and this flag (0x2) is set, displays the Enable and Mask registers for each bank.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!gpiokd.gpioext

Article • 04/03/2024

The **!gpiokd.gpioext** extension displays information about a GPIO controller.

dbgcmd

```
!gpiokd.gpioext ExtensionAddress [Flags]
```

Parameters

ExtensionAddress

Address of the [_DEVICE_EXTENSION](#) structure that represents the GPIO controller.

Flags

Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

[+] Expand table

Flag	Description
0x1	Displays the pin table for each bank.
0x2	If bit 0 (0x1) is set and this flag (0x2) is set, the display includes the Enable and Mask registers for each bank.
0x4	If bit 0 (0x1) is set and this flag (0x4) is set, the display includes unconfigured pins.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!gpiokd.pininfo

Article • 04/03/2024

The `!gpiokd.pininfo` extension displays information about a specified GPIO pin.

```
dbgcmd
```

```
!gpiokd.pininfo PinAddress
```

Parameters

PinAddress

Address of the [_GPIO_PIN_INFORMATION_ENTRY](#) data structure that represents the pin.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!gpiokd.pinisrvec

Article • 04/03/2024

The `!gpiokd.pinisrvec` extension displays Interrupt Service Routine (ISR) vector information for a specified pin.

dbgcmd

```
!gpiokd.bankinfo PinAddress
```

Parameters

PinAddress

Address of the [_GPIO_PIN_INFORMATION_ENTRY](#) data structure that represents the pin.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!gpiokd.pintable

Article • 04/03/2024

The **!gpiokd.pintable** extension displays information about an array of GPIO pins.

dbgcmd

```
!gpiokd.pintable PinBase PinCount [Flags]
```

Parameters

PinBase

Address of an array of [_GPIO_PIN_INFORMATION_ENTRY](#) structures.

PinCount

The number of pins to display.

Flags

Flags that specify which information is displayed. This parameter is a bitwise OR of one or more of the following flags.

[+] Expand table

Flag	Description
0x1	Not used by this command.
0x2	Not used by this command.
0x4	The display includes unconfigured pins.

DLL

Gpiokd.dll

See also

[GPIO Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

USB 3.0 Extensions

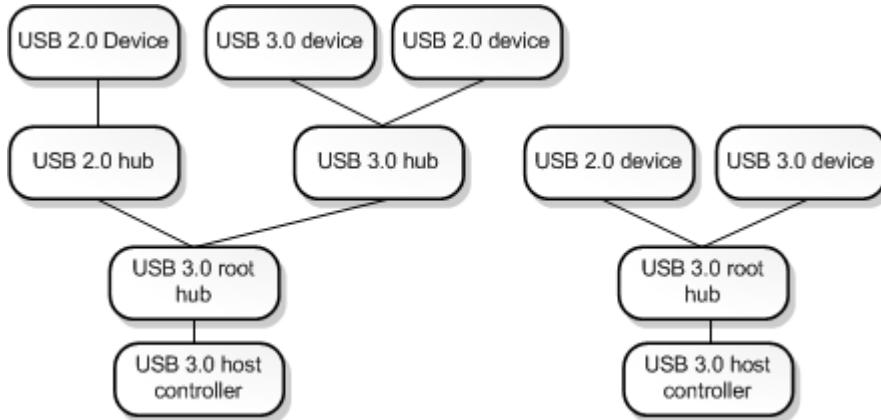
Article • 10/25/2023

This section describes the USB 3.0 debugger extension commands. These commands display information from data structures maintained by three drivers in the USB 3.0 stack: the USB 3.0 hub driver, the USB host controller extension driver, and the USB 3.0 host controller driver. For more information about these three drivers, see [USB host-side drivers in Windows](#). For an explanation of the data structures used by the drivers in the USB 3.0 stack, see [USB 3.0 Data Structures](#) and Part 2 of the [USB Debugging Innovations in Windows 8](#) video.

The USB 3.0 debugger extension commands are implemented in Usb3kd.dll. To load the Usb3kd commands, enter `.load usb3kd.dll` in the debugger.

USB 3.0 Tree

The USB 3.0 tree contains all USB 3.0 host controllers and all hubs and devices that are connected to USB 3.0 host controllers. The following diagram shows an example of a USB 3.0 tree.



The tree shown in the diagram has two USB 3.0 host controllers. Notice that not every device shown in the diagram is a USB 3.0 device. But all of the devices shown (including the hubs) are part of the USB 3.0 tree, because each device is on a branch that originates at a USB 3.0 host controller.

You can think of the diagram as two trees, one for each host controller. However, when we use the term *USB 3.0 tree*, we are referring to the set of all USB 3.0 host controllers along with their connected hubs and devices.

Getting started with USB 3.0 debugging

To start debugging a USB 3.0 issue, enter the [!usb_tree](#) command. The [!usb_tree](#) command displays a list of commands and addresses that you can use to investigate host controllers, hubs, ports, devices, endpoints, and other elements of the USB 3.0 tree.

Hub commands

The following extension commands display information about USB 3.0 hubs, devices, and ports. The displayed information is based on data structures maintained by the USB 3.0 hub driver.

- [!usb3kd.usb_tree](#)
- [!usb3kd.hub_info](#)
- [!usb3kd.hub_info_from_fdo](#)
- [!usb3kd.device_info](#)
- [!usb3kd.device_info_from_pdo](#)
- [!usb3kd.port_info](#)

UCX commands

The following extension commands display information about USB 3.0 host controllers, devices, and ports. The displayed information is based on data structures maintained by the USB host controller extension driver.

- [!usb3kd.uxc_controller_list](#)
- [!usb3kd.uxc_controller](#)
- [!usb3kd.uxc_device](#)
- [!usb3kd.uxc_endpoint](#)

Host controller commands

The following extension commands display information from data structures maintained by the USB 3.0 host controller driver.

- [!usb3kd.xhci_dumpall](#)
- [!usb3kd.xhci_capability](#)
- [!usb3kd.xhci_commandring](#)
- [!usb3kd.xhci_deviceslots](#)
- [!usb3kd.xhci_eventring](#)
- [!usb3kd.xhci_registers](#)
- [!usb3kd.xhci_resourceusage](#)
- [!usb3kd.xhci_trb](#)

- [!usb3kd.xhci_transferring](#)
- [!usb3kd.xhci_findowner](#)

Miscellaneous commands

- [!usb3kd.usbdstatus](#)
- [!usb3kd.urb](#)

See also

[RCDRKD Extensions](#)

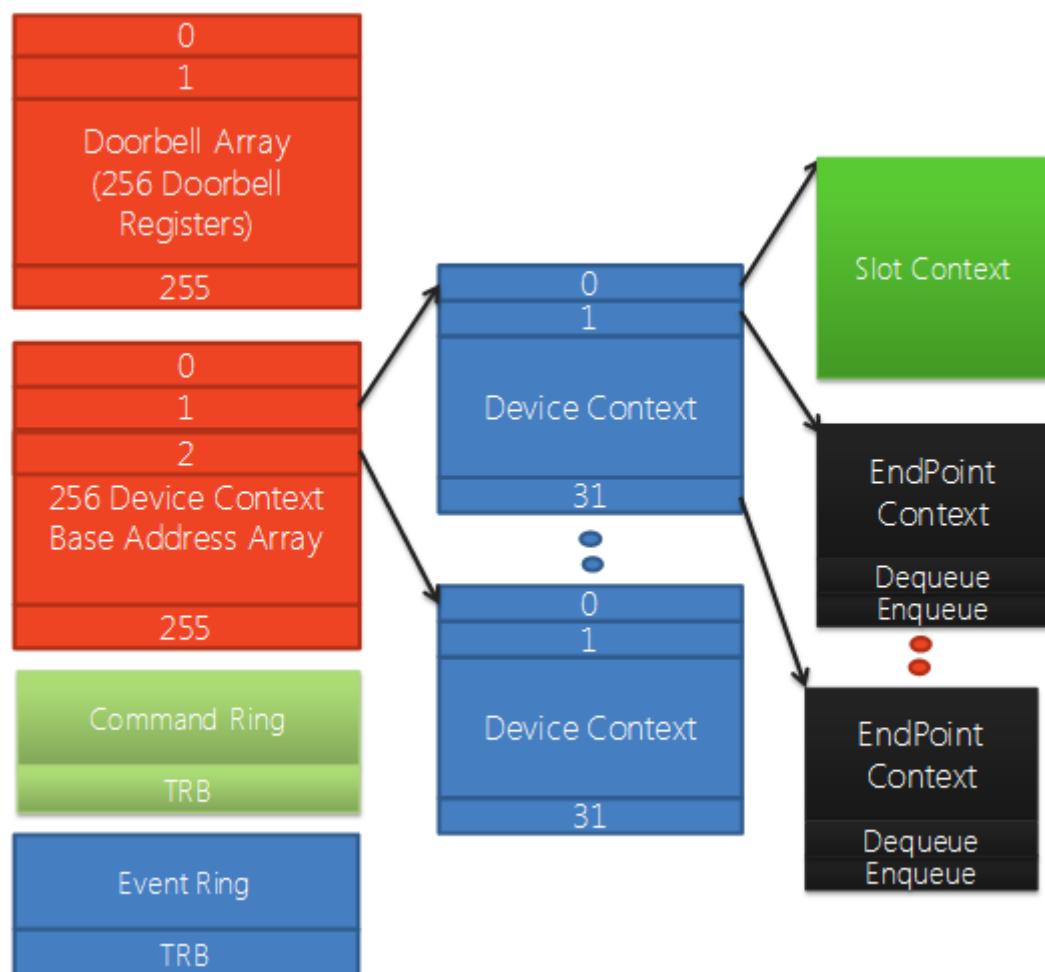
USB 3.0 Data Structures

Article • 10/25/2023

This topic describes the data structures used by the USB 3.0 host controller driver. Understanding these data structures will help you use the [USB 3.0](#) and [RCDRKD](#) debugger extension commands effectively. The data structures presented here have names that are consistent with the [USB 3.0 specification](#). Familiarity with the USB 3.0 specification will further enhance your ability to use the extension commands to debug USB 3.0 drivers.

The USB 3.0 host controller driver is part of the USB 3.0 core driver stack. For more information, see [USB Driver Stack Architecture](#).

Each USB 3.0 host controller can have up to 255 devices, and each device can have up to 31 endpoints. The following diagram shows some of the data structures that represent one host controller and the connected devices.



Device Context Base Array

The Device Context Base Array is an array of pointers to Device Context structures. There is one Device Context structure for each device connected to the host controller. Elements 1 through 255 point to Device Context structures. Element 0 points to a context structure for the host controller.

Device Context and Slot Context

A Device Context structure holds an array of pointers to Endpoint Context structures. There is one Endpoint Context structure for each endpoint on the device. Elements 1 through 31 point to Endpoint Context structures. Element 0 points to a Slot Context structure, which holds context information for the device.

Command Ring

The Command Ring is used by software to pass commands to the host controller. Some of these commands are directed at the host controller, and some are directed at particular devices connected to the host controller.

Event Ring

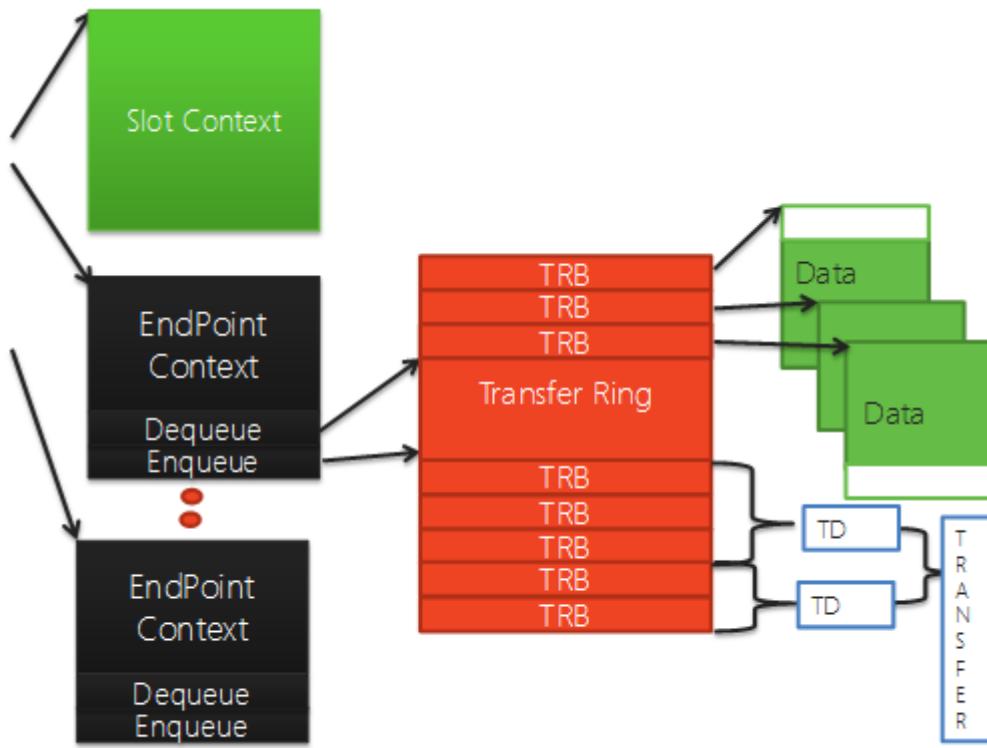
The Event Ring is used by the host controller to pass events to software. That is, the Event Ring is a structure that the host controller uses to inform drivers that an action has completed.

Doorbell Register Array

The Doorbell Register Array is an array of doorbell registers, one for each device connected to the host controller. Elements 1 through 255 are doorbell registers. Element 0 indicates whether there is a pending command in the Command Ring.

Software notifies the host controller that it has device-related or endpoint-related work to perform by writing context information into the doorbell register for the device.

The following diagram continues to the right of the preceding diagram. It shows additional data structures that represent a single endpoint.



Transfer Ring

Each endpoint has one or more Transfer Rings. A Transfer Ring is an array of Transfer Request Blocks (TRBs). Each TRB points to a block of contiguous data (up to 64 KB) that will be transferred between hardware and memory as a single unit.

When the USB 3.0 core stack receives a transfer request from a USB client driver, it identifies the Endpoint Context for the transfer, and then breaks the transfer request into one or more Transfer Descriptors (TDs). Each TD contains one or more TRBs.

Endpoint Context

An Endpoint Context structure holds context information for a single endpoint. It also has **Dequeue** and **Enqueue** members, which are used to track where TRBs are being consumed by the hardware and where TRBs are being added by software.

See also

[USB Debugging Innovations in Windows 8](#)

!usb3kd.help

Article • 10/25/2023

The **!usb3kd.help** command displays help for the USB 3 debugger extension commands.

DLL

Usb3kd.dll

See also

[USB 3.0 Extensions](#)

!usb3kd.device_info

Article • 10/25/2023

The **!usb3kd.device_info** command displays information about a USB device in the [USB 3.0 tree](#).

```
dbgcmd
```

```
!usb3kd.device_info DeviceContext
```

Parameters

DeviceContext

Address of the _DEVICE_CONTEXT structure that represents the device.

DLL

Usb3kd.dll

Remarks

!device_info and [**!ucx_device**](#) both display information about a device, but the information displayed is different. The output of **!device_info** is from the point of view of the USB 3.0 hub driver, and the output of [**!ucx_device**](#) is from the point of view of the USB host controller extension driver. For example, the **!device_info** output includes information about configuration and interface descriptors, and [**!ucx_device**](#) output includes information about endpoints.

Examples

You can obtain the address of the device context structure by looking at the output of the [**!usb_tree**](#) command. In the following example, the address of the device context structure is 0xfffffa8005abd0c0.

```
dbgcmd
```

```
3: kd> !usb_tree
```

```
## Dumping HUB Tree - !drvObj 0xfffffa800597f770
```

```
-----
```

```

Topology
-----
1) !xhci_info 0xfffffa80051d1940 ... - PCI: VendorId ...
   !hub_info 0xfffffa8005ad92d0 (ROOT)
   ...
## Enumerated Device List
-----
...
2) !device_info 0xfffffa8005abd0c0, !devstack fffffa80059c3800
   Current Device State: ConfiguredInD0
   Desc: ... USB Flash Drive
   ...

```

Now you can pass the address of the device context to the `!device_info` command.

```

dbgcmd

3: kd> !device_info 0xfffffa8005abd0c0

## Dumping Device Information fffffa8005abd0c0
-----
dt USBHUB3!_DEVICE_CONTEXT 0xfffffa8005abd0c0
dt USBHUB3!_HUB_PDO_CONTEXT 0xfffffa8005b118d0
!idle_info 0xfffffa8005b11920 (dt USBHUB3!_ISM_CONTEXT 0xfffffa8005b11920)
Parent !hub_info 0xfffffa8005ad92d0 (dt USBHUB3!_HUB_FDO_CONTEXT
0xfffffa8005ad92d0)
!port_info 0xfffffa8005abe0c0 (dt USBHUB3!_PORT_CONTEXT 0xfffffa8005abe0c0)
!ucx_device 0xfffffa8005992840 !xhci_deviceslots 0xfffffa80051d1940 1

LPMState: U1IsEnabledForUpStreamPort U2IsEnabledForUpStreamPort U1Timeout:
38, U2Timeout: 3
DeviceFlags: HasContainerId NoBOSContainerId Removable HasSerialNumber
MsOsDescriptorNotSupported NoWakeUpSupport DeviceIsSuperSpeedCapable
DeviceHackFlags: WillDisableOnSoftRemove SkipSetIsochDelay
WillResetOnResumeS0 DisableOnSoftRemove

Descriptors:
dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa80053f9250
dt _USB_INTERFACE_DESCRIPTOR 0xfffffa80053f9259
ProductId: ... Flash Drive
DeviceDescriptor: VID ... PID ... REV 0a00, Class: (0)(0) BcdUsb: 0300

UcxRequest: !wdfrequest 0x57ffa662948,
ControlRequest: !wdfrequest 0x57ffa667798, !irp 0xfffffa8005997650 !urb
0xfffffa8005abd1c0, NumberOfBytes 0
Device working at SuperSpeed
Current Device State: ConfiguredInD0

Device State History: <Event> NewState (<Operation>(),..) :

[16] <Yes> ConfiguredInD0

```

```
...
Device Event History:
[10] TransferSuccess
...
```

See also

[USB 3.0 Extensions](#)

[`!usb3kd.device_info_from_pdo`](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.device_info_from_pdo

Article • 10/25/2023

The **!usb3kd.device_info_from_pdo** command displays information about a USB device in the [USB 3.0 tree](#).

dbgcmd

```
!usb3kd.device_info_from_pdo DeviceObject
```

Parameters

DeviceObject

Address of the physical device object (PDO) of a USB device or hub.

DLL

Usb3kd.dll

Remarks

!device_info_from_pdo and **!lucx_device** both display information about a device, but the information displayed is different. The output of **!device_info_from_pdo** is from the point of view of the USB 3.0 hub driver, and the output of **!lucx_device** is from the point of view of the USB host controller extension driver. For example, the **!device_info_from_pdo** output includes information about configuration and interface descriptors, and **!lucx_device** output includes information about endpoints.

Examples

You can get the address of the PDO from the output of [!usb_tree](#) or from a variety of other debugger commands. For example, the [!devnode](#) command displays the addresses of PDOs. In the following example, the USBSTOR device node is the direct child of the USBHUB3 node. The address of the PDO for the USBSTOR node is 0xfffffa80059c3800.

dbgcmd

```

3: kd> !devnode 0 1 usbhub3

Dumping IopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8005981730 for PDO 0xfffffa8004ffc550
  InstancePath is "USB\ROOT_HUB30\5&11db9684&0&0"
  ServiceName is "USBHUB3"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)

DevNode 0xfffffa8005a546a0 for PDO 0xfffffa80059c3800
  InstancePath is "USB\VID_125F&PID_312A\0902100000000000342192873"
  ServiceName is "USBSTOR"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)

DevNode 0xfffffa8005a09730 for PDO 0xfffffa8005b3a650
  InstancePath is "USBSTOR\Disk&Ven ..."
  ServiceName is "disk"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)

```

Now you can pass the address of the PDO to the `!usb3kd.device_info_from_pdo` command.

```

dbgcmd

3: kd> !device_info_from_pdo 0xfffffa80059c3800

## Dumping Device Information fffffa80059c3800
-----
!devobj 0xfffffa8004ffc550 (Root HUB)
!device_info 0xfffffa8005abd0c0 (dt usbhub3!_DEVICE_CONTEXT
0xfffffa8005abd0c0)
dt USBHUB3!_DEVICE_CONTEXT 0xfffffa8005abd0c0
dt USBHUB3!_HUB_PDO_CONTEXT 0xfffffa8005b118d0
!idle_info 0xfffffa8005b11920 (dt USBHUB3!_ISM_CONTEXT 0xfffffa8005b11920)
Parent !hub_info 0xfffffa8005ad92d0 (dt USBHUB3!_HUB_FDO_CONTEXT
0xfffffa8005ad92d0)
!port_info 0xfffffa8005abe0c0 (dt USBHUB3!_PORT_CONTEXT 0xfffffa8005abe0c0)
!ucx_device 0xfffffa8005992840 !xhci_deviceslots 0xfffffa80051d1940 1

LPMState: U1IsEnabledForUpStreamPort U2IsEnabledForUpStreamPort U1Timeout:
38, U2Timeout: 3
DeviceFlags: HasContainerId NoBOSContainerId Removable HasSerialNumber
MsOsDescriptorNotSupported NoWakeUpSupport DeviceIsSuperSpeedCapable
DeviceHackFlags: WillDisableOnSoftRemove SkipSetIsochDelay
WillResetOnResumeS0 DisableOnSoftRemove

Descriptors:
dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa80053f9250
dt _USB_INTERFACE_DESCRIPTOR 0xfffffa80053f9259
ProductId: ...

```

```
DeviceDescriptor: VID ...

UcxRequest: !wdfrequest 0x57ffa662948,
ControlRequest: !wdfrequest 0x57ffa667798, !irp 0xfffffa8005997650 !urb
0xfffffa8005abd1c0, NumberOfBytes 0
Device working at SuperSpeed
Current Device State: ConfiguredInD0

Device State History: <Event> NewState (<Operation>(),...)
[16] <Yes> ConfiguredInD0
...
Device Event History:
[10] TransferSuccess
...
```

The following example shows some of the output of the [!usb_tree](#) command. You can see the address of the PDO of one of the child device nodes as the argument to the [!devstack](#) command. ([!devstack fffffa80059c3800](#))

```
dbgcmd

3: kd> !usb_tree

## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-----
Topology
-----
1) !xhci_info 0xfffffa80051d1940 ... - PCI: VendorId ...
   !hub_info 0xfffffa8005ad92d0 (ROOT)
      !port_info 0xfffffa8005a5ca80 !device_info 0xfffffa8005b410c0 Desc:
<none> Speed: High
      ...
## Enumerated Device List
-----
...
2) !device_info 0xfffffa8005abd0c0, !devstack fffffa80059c3800
   Current Device State: ConfiguredInD0
   Desc: ... Flash Drive
   USB\VID_...
   !ucx_device 0xfffffa8005992840 !xhci_deviceslots 0xfffffa80051d1940 1
```

See also

[USB 3.0 Extensions](#)

[!usb3kd.device_info](#)

Universal Serial Bus (USB) Drivers

!usb3kd.ds

Article • 10/25/2023

The **!usb3kd.ds** extension is a toggle command that sets the debugger context to debug the DSF host driver.

```
dbgcmd
```

```
!usb3kd.ds
```

DLL

Usb3kd.dll

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.hub_info

Article • 10/25/2023

The [!usb3kd.device_info](#) command displays information about a hub in the [USB 3.0 tree](#).

```
dbgcmd
```

```
!usb3kd.hub_info DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the hub's functional device object (FDO).

DLL

Usb3kd.dll

Examples

To obtain the address of the device extension, look at the output of the [!usb_tree](#) command. In the following example, the address of the device extension for the root hub is 0xfffffa8005ad92d0.

```
dbgcmd
```

```
3: kd> !usb_tree
## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-----
Topology
-----
1)  !xhci_info 0xfffffa80051d1940 ... - PCI: VendorId 0x1033 ...
    !hub_info 0xfffffa8005ad92d0 (ROOT)
        !port_info 0xfffffa8005a5ca80 !device_info 0xfffffa8005b410c0 Desc:
<none> Speed: High
    ...
```

Now you can pass the address of the device extension to the [!hub_info](#) command.

```
dbgcmd
```

```
3: kd> !hub_info ffffffa8005ad92d0

## Dumping HUB Information ffffffa8005ad92d0
-----
dt USBHUB3!_HUB_FDO_CONTEXT 0xfffffa8005ad92d0
!rcdrlogdump usbhub3 -a 0xfffffa8005ad8010
Capabilities: Initialized, Configured, HasOverCurrentProtection, ...

Total number of ports: 4, HUB depth is 0
Number of 3.0 ports: 2 (Range 1 - 2)
Number of 2.0 ports: 2 (Range 3 - 4)

Descriptors:
dt _USB_HUB_DESCRIPTOR 0xfffffa8005ad9728
dt _USB_30_HUB_DESCRIPTOR 0xfffffa8005ad9728
dt _USB_DEVICE_DESCRIPTOR 0xfffffa8005ad92d0
dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa8005ad9770

List of PortContext: 4
[1] !port_info 0xfffffa8005a5ca80      !device_info 0xfffffa8005b410c0
    Last Port Status(2.0): Connected Enabled Powered
HighSpeedDeviceAttached
    Last Port Change: <none>
    Current Port(2.0) State:
ConnectedEnabled.WaitingForPortChangeEvent
    Current Device State: ConfiguredInD0
    ...
    Current Hub State: ConfiguredWithIntTransfer

Hub State History: <Event> NewState (<Operation>(),...) :

[43] <OperationSuccess> ConfiguredWithIntTransfer
...
Hub Event History:
[ 0] PortEnableInterruptTransfer
...
```

See also

[USB 3.0 Extensions](#)

[`!usb3kd.hub_info_from_fdo`](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.hub_info_from_fdo

Article • 10/25/2023

The **!usb3kd.hub_info_from_fdo** command displays information about a hub in the [USB 3.0 tree](#).

```
dbgcmd
```

```
!usb3kd.hub_info_from_fdo DeviceObject
```

Parameters

DeviceObject

Address of the functional device object (FDO) that represents the hub.

DLL

Usb3kd.dll

Examples

You can get the address of the FDO from the output of [!usb_tree](#) or from a variety of other debugger commands. For example, the [!devstack](#) command displays the address of the FDO. In the following example, the address of the FDO is ffffffa800597a660.

```
dbgcmd
```

```
3: kd> !devnode 0 1 ushub3
Dumping IopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8005981730 for PDO 0xfffffa8004ffc550
...
3: kd> !devstack 0xfffffa8004ffc550
  !DevObj   !DrvObj          !DevExt   ObjectName
  ffffffa800597a660  \Driver\USBHUB3    ffffffa8005ad92d0
> ffffffa8004ffc550  \Driver\USBXHCI    ffffffa8005251d00  USBPDO-0
...
```

Now you can pass the address of the FDO to the **!usb3kd.hub_info_from_fdo** command.

```
dbgcmd
```

```

3: kd> !hub_info_from_fdo ffffffa800597a660

## Dumping HUB Information ffffffa800597a660
-----
dt USBHUB3!_HUB_FDO_CONTEXT 0xfffffa8005ad92d0
!rcdrlogdump usbhub3 -a 0xfffffa8005ad8010
Capabilities: Initialized, Configured, HasOverCurrentProtection,
SelectiveSuspendSupportedByParentStack, CannotWakeOnConnect, NotArmedForWake

Total number of ports: 4, HUB depth is 0
Number of 3.0 ports: 2 (Range 1 - 2)
Number of 2.0 ports: 2 (Range 3 - 4)

Descriptors:
    dt _USB_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_30_HUB_DESCRIPTOR 0xfffffa8005ad9728
    dt _USB_DEVICE_DESCRIPTOR 0xfffffa8005ad92d0
    dt _USB_CONFIGURATION_DESCRIPTOR 0xfffffa8005ad9770

List of PortContext: 4
[1] !port_info 0xfffffa8005a5ca80      !device_info 0xfffffa8005b410c0
    Last Port Status(2.0): Connected Enabled Powered
HighSpeedDeviceAttached
    Last Port Change: <none>
    Current Port(2.0) State:
ConnectedEnabled.WaitingForPortChangeEvent
    Current Device State: ConfiguredInD0
    ...
Current Hub State: ConfiguredWithIntTransfer

Hub State History: <Event> NewState (<Operation>(),...) :
[43] <OperationSuccess> ConfiguredWithIntTransfer
    ...

Hub Event History:
[ 0] PortEnableInterruptTransfer
    ...

```

See also

[USB 3.0 Extensions](#)

[**!usb3kd.hub_info**](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.port_info

Article • 10/25/2023

The [!usb3kd.port_info](#) command displays information about a USB port in the [USB 3.0 tree](#).

```
dbgcmd
```

```
!usb3kd.port_info PortContext
```

Parameters

PortContext

Address of a `_PORT_CONTEXT` structure.

DLL

Usb3kd.dll

Examples

To obtain the address of the port context, look at the output of the [!usb_tree](#) command. In the following example, the address of a port context is `0xfffffa8005abe0c0`.

```
dbgcmd
```

```
3: kd> !usb_tree
## Dumping HUB Tree - !drvObj 0xfffffa800597f770
-----
Topology
-----
1) !xhci_info 0xfffffa80051d1940 ... - PCI: VendorId ...
   !hub_info 0xfffffa8005ad92d0 (ROOT)
   ...
   !port_info 0xfffffa8005abe0c0 !device_info 0xfffffa8005abd0c0 Desc:
... USB Flash Drive Speed: Super
```

Now you can pass the address of the port context to the [!port_info](#) command.

```
dbgcmd
```

```
3: kd> !port_info 0xfffffa8005abe0c0

## Dumping Port Context 0xfffffa8005abe0c0
-----
dt USBHUB3!_PORT_CONTEXT 0xfffffa8005abe0c0
!hub_info 0xfffffa8005ad92d0 (dt _HUB_FDO_CONTEXT 0xfffffa8005ad92d0)
!device_info 0xfffffa8005abd0c0 (dt _DEVICE_CONTEXT 0xfffffa8005abd0c0)
!rcdrlogdump usbhub3 -a 0xfffffa8005abf6b0

PortNumber: 2
Last Port Status(3.0): Connected Enabled Powered
Last Port Change: <none>

CurrentPortEvent: PsmEventPortConnectChange
Current Port(3.0) State: ConnectedEnabled.WaitingForPortChangeEvent

Port(3.0) State History: <Event> NewState (<Operation>(),...) :

[34] <Push> WaitingForPortChangeEvent
...
Port Event History:
[ 8] TransferSuccess
...
```

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.uxc_device

Article • 10/25/2023

The **!usb3kd.uxc_device** extension displays information about a USB device in the [USB 3.0 tree](#). The display is based on data structures maintained by the USB host controller extension driver (*UcxVersion.sys*).

dbgcmd

```
!usb3kd.uxc_device UcxUsbDevicePrivContext
```

Parameters

UcxUsbDevicePrivContext

Address of the `_UCXUSBDEVICE_PRIVCONTEXT` structure that represents the device.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (*UcxVersion.sys*) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output of the **!ucx_device** command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see [USB Driver Stack Architecture](#).

!ucx_device and **!device_info** both display information about a device, but the information displayed is different. The output of **!ucx_device** is from the point of view of the USB host controller extension driver, and the output of **!device_info** is from the point of view of the USB 3.0 hub driver. For example, the **!ucx_device** output includes information about endpoints, and the **!device_info** output includes information about configuration and interface descriptors.

Examples

To obtain the address of the UCX USB device private context, look at the output of the [!ucx_controller_list](#) command. In the following example, the address of the private context for the second device is 0xfffffa8005bd9680.

```
dbgcmd

3: 3: kd> !ucx_controller_list

## Dumping List of UCX controller objects
-----
[1] !ucx_controller 0xfffffa80052da050 (dt
    ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        ...
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        ...

```

Now you can pass the address of the UCX USB private context to the [!ucx_device](#) command.

```
dbgcmd

3: kd> !ucx_device 0xfffffa8005bd9680

## Dumping Ucx USB Device Information fffffa8005bd9680
-----
dt ucx01000!_UCXUSBDEVICE_PRIVCONTEXT 0xfffffa8005bd9680
!ucx_controller 0xfffffa80052da050
ParentHub: !wdfhandle 0x57ffacbcce78
DefaultEndpoint: !ucx_endpoint 0xfffffa8005be0550
ListOfEndpoints:
    .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8003686820 [Blk In ], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8005be0550 [Control], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8003695580 [Blk In ], UcxEndpointStateStale
    .!ucx_endpoint 0xfffffa80036a20c0 [Blk Out], UcxEndpointStateStale

EventCallbacks:
    EvtUsbDeviceEndpointsConfigure: (0xfffff880044d1164)
USBXHCI!UsbDevice_UcxEvtEndpointsConfigure
    EvtUsbDeviceEnable: (0xfffff880044cffac) USBXHCI!UsbDevice_UcxEvtEnable
    EvtUsbDeviceDisable: (0xfffff880044d1cbc)
USBXHCI!UsbDevice_UcxEvtDisable
    EvtUsbDeviceReset: (0xfffff880044d2178) USBXHCI!UsbDevice_UcxEvtReset
    EvtUsbDeviceAddress: (0xfffff880044d0934)
USBXHCI!UsbDevice_UcxEvtAddress
    EvtUsbDeviceUpdate: (0xfffff880044d0c80) USBXHCI!UsbDevice_UcxEvtUpdate
    EvtUsbDeviceDefaultEndpointAdd: (0xfffff880044ede1c)
USBXHCI!Endpoint_UcxEvtUsbDeviceDefaultEndpointAdd
```

```
EvtUsbDeviceEndpointAdd: (0xfffff880044edfc8)
USBXHCI!Endpoint_UcxEvtUsbDeviceEndpointAdd
```

See also

[USB 3.0 Extensions](#)

[!usb3kd.ucs_controller_list](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.uxc_endpoint

Article • 10/25/2023

The **!usb3kd.uxc_endpoint** command displays information about an endpoint on a USB device in the [USB 3.0 tree](#). The display is based on data structures maintained by the USB host controller extension driver (*UcxVersion.sys*).

```
dbgcmd
```

```
!usb3kd.uxc_endpoint UcxEndpointPrivContext
```

Parameters

UcxEndpointPrivContext

Address of the `_UCXENDPOINT_PRIVCONTEXT` structure that represents the endpoint.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (*UcxVersion.sys*) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output of the **!ucx_endpoint** command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see [USB Driver Stack Architecture](#).

Examples

To obtain the address of the UCX endpoint private context, look at the output of the **!ucx_controller_list** command. In the following example, the address of the private context for the first endpoint on the second device is `0xfffffa8003694860`.

```
dbgcmd
```

```
3: kd> !ucx_controller_list
```

```

## Dumping List of UCX controller objects
-----
[1] !ucx_controller 0xfffffa80052da050 (dt
    ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa80053405d0 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005a3f710 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005bbe4e0 [Blk Out], UcxEndpointStateStale
        .!ucx_endpoint 0xfffffa8005ac4810 [Blk In ], UcxEndpointStateStale
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8003686820 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005be0550 [Control], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8003695580 [Blk In ], UcxEndpointStateStale
        .!ucx_endpoint 0xfffffa80036a20c0 [Blk Out], UcxEndpointStateStale

```

Now you can pass the address of the UCX endpoint private context to the `!ucx_endpoint` command.

```

dbgcmd

3: kd> !ucx_endpoint 0xfffffa8003694860

## Dumping Ucx USB Endpoint Information fffffa8003694860
-----
dt ucx01000!_UCXENDPOINT_PRIVCONTEXT 0xfffffa8003694860
[Blk Out], UcxEndpointStateEnabled, MaxTransferSize: 4194304
Endpoint Address: 0x02
Endpoint Queue: !wdfqueue 0x57ffc969888

UcxEndpoint State History: <Event> NewState
[ 3] <UcxEndpointEventOperationSuccess> UcxEndpointStateEnabled
[ 2] <UcxEndpointEventYes> UcxEndpointStateCompletingPendingOperation1
[ 1] <UcxEndpointEventEnableComplete> UcxEndpointStateIsAbleToStart2
[ 0] <SmEngineEventStart> UcxEndpointStateCreated

UcxEndpoint Event History:
[ 1] UcxEndpointEventEnableComplete
[ 0] SmEngineEventStart

EventCallbacks:
    EvtEndpointPurge: (0xfffff880044ba6e8)
    USBXHCI!Endpoint_UcxEvtEndpointPurge
        EvtEndpointAbort: (0xfffff880044ba94c)
    USBXHCI!Endpoint_UcxEvtEndpointAbort
        EvtEndpointReset: (0xfffff880044bb854)
    USBXHCI!Endpoint_UcxEvtEndpointReset

```

See also

USB 3.0 Extensions

[**!usb3kd.ucx_controller_list**](#)

Universal Serial Bus (USB) Drivers

!usb3kd.uxc_controller

Article • 10/25/2023

The [!usb3kd.uxc_controller](#) command displays information about a USB 3.0 host controller. The display is based on data structures maintained by the USB host controller extension driver (*UcxVersion.sys*).

dbgcmd

```
!usb3kd.uxc_controller UcxControllerPrivContext
```

Parameters

UcxControllerPrivContext

Address of the `_UCXCONTROLLER_PRIVCONTEXT` structure that represents the controller.

DLL

Usb3kd.dll

Remarks

The USB host controller extension driver (*UcxVersion.sys*) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The output of the [!ucx_controller](#) command is based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see [USB Driver Stack Architecture](#).

Examples

To obtain the address of the UCX controller private context, look at the output of the [!ucx_controller_list](#) command. In the following example, the address of the private context is 0xfffffa80052da050.

dbgcmd

```
3: kd> !ucx_controller_list

## Dumping List of UCX controller objects
-----
[1] !ucx_controller 0xfffffa80052da050 (dt
ucx01000!_UCXCONTROLLER_PRIVCONTEXT fffffa80052da050)
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        ...
    !ucx_device 0xfffffa8005bd9680
        .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
        ...

```

Now you can pass the address of the UCX controller private context to the [!ucx_controller](#) command.

```
dbgcmd

3: kd> !ucx_controller 0xfffffa80052da050

## Dumping Ucx Controller Information fffffa80052da050
-----
dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT 0xfffffa80052da050
Parent Device: !wdfdevice 0x57ffac91fd8
Controller Queues:
    Default          : !wdfqueue 0x57ffacc5fd8
    Address'0'Ownership : !wdfqueue 0x57ffad5ad88
    DeviceManagement   : !wdfqueue 0x57ffacd6fd8
    ... pend on Ctrl Reset: !wdfqueue 0x57ffad48fd8

Controller Reset State History: <Event> NewState
    [ 2] <ControllerResetEventOperationSuccess>
ControllerResetStateRHPdoInD0
    [ 1] <ControllerResetEventRHPdoEnteredD0>
ControllerResetStateStopBlockingResetComplete1
    [ 0] <SmEngineEventStart> ControllerResetStateRhPdoInDx

Controller Reset Event History:
    [ 1] ControllerResetEventRHPdoEnteredD0
    [ 0] SmEngineEventStart

Root Hub PDO: !wdfdevice 0x57ffaf4daa8
Number of 2.0 Ports: 2
Number of 3.0 Ports: 2
RootHub Control !wdfqueue 0x57ffacb4798
RootHub Interrupt !wdfqueue 0x57ffad033f8, pending !wdfequest 0x57ffa5fe998

Device Tree:
    !ucx_device 0xfffffa8005a41840
        .!ucx_endpoint 0xfffffa800533f3d0 [Blk In ], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa80053405d0 [Blk Out], UcxEndpointStateEnabled
        .!ucx_endpoint 0xfffffa8005a3f710 [Control], UcxEndpointStateEnabled
```

```
.!ucx_endpoint 0xfffffa8005bbe4e0 [Blk Out], UcxEndpointStateStale
.!ucx_endpoint 0xfffffa8005ac4810 [Blk In ], UcxEndpointStateStale
!ucx_device 0xfffffa8005bd9680
.!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
.!ucx_endpoint 0xfffffa8003686820 [Blk In ], UcxEndpointStateEnabled
.!ucx_endpoint 0xfffffa8005be0550 [Control], UcxEndpointStateEnabled
.!ucx_endpoint 0xfffffa8003695580 [Blk In ], UcxEndpointStateStale
.!ucx_endpoint 0xfffffa80036a20c0 [Blk Out], UcxEndpointStateStale
```

See also

[USB 3.0 Extensions](#)

[**!usb3kd.ucs_controller_list**](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.uxc_controller_list

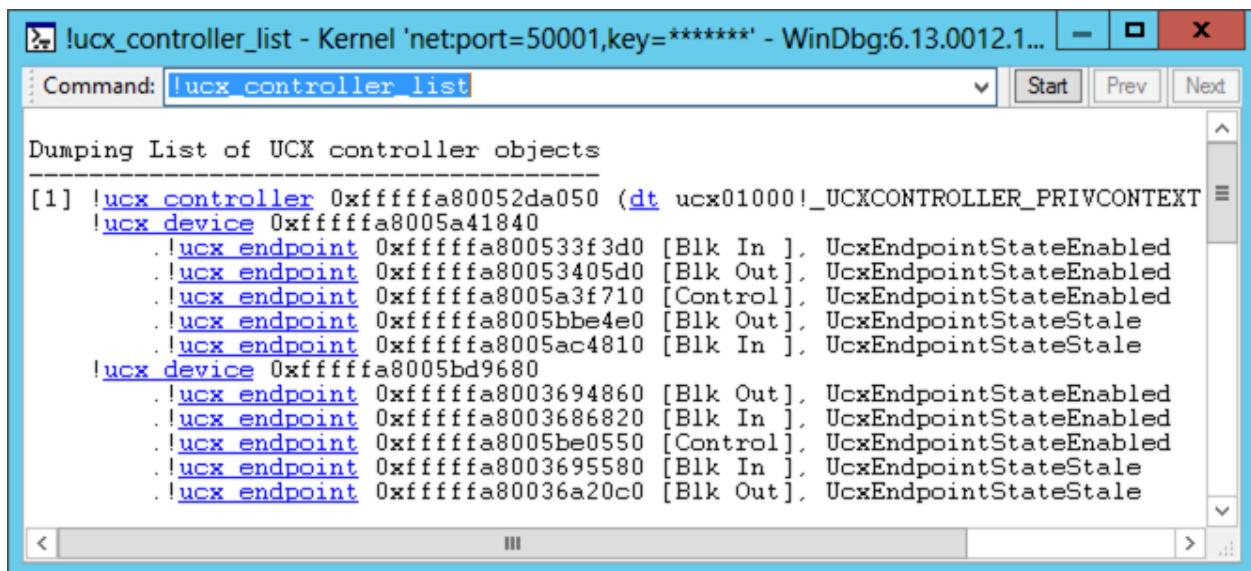
Article • 10/25/2023

The [!usb3kd.uxc_controller_list](#) command displays information about all USB 3.0 host controllers on the computer. The display is based on data structures maintained by the USB host controller extension driver (*UcxVersion.sys*).

```
dbgcmd
!usb3kd.uxc_controller_list
```

Examples

The following screen shot show the output of the [!ucx_controller_list](#) command.



```
Dumping List of UCX controller objects
[1] !ucx_controller 0xfffffa80052da050 (dt ucx01000!_UCXCONTROLLER_PRIVCONTEXT
  !ucx_device 0xfffffa8005a41840
    .!ucx_endpoint 0xfffffa800533f3d0 [Blk In], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa80053405d0 [Blk Out], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8005a3f710 [Control], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8005bbe4e0 [Blk Out], UcxEndpointStateStale
    .!ucx_endpoint 0xfffffa8005ac4810 [Blk In], UcxEndpointStateStale
  !ucx_device 0xfffffa8005bd9680
    .!ucx_endpoint 0xfffffa8003694860 [Blk Out], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8003686820 [Blk In], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8005be0550 [Control], UcxEndpointStateEnabled
    .!ucx_endpoint 0xfffffa8003695580 [Blk In], UcxEndpointStateStale
    .!ucx_endpoint 0xfffffa80036a20c0 [Blk Out], UcxEndpointStateStale
```

The output shows that there is one USB 3.0 host controller, which is represented by the line that begins with [!ucx_controller](#). You can see that two devices are connected to the controller and that each device has four endpoints.

The output uses [Using Debugger Markup Language \(DML\)](#) to provide links. The links execute commands that give detailed information about individual devices or endpoints. For example, you could get detailed information about an endpoint by clicking one of the [!ucx_endpoint](#) links. As an alternative to clicking a link, you can enter a command. For example, to see information about the first endpoint of the second device, you could enter the command `!ucx_endpoint 0xfffffa8003694860`.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The [!lucx_controller_list](#) command is the parent command for this set of commands.

- [!lucx_controller](#)
- [!lucx_device](#)
- [!lucx_endpoint](#)

The USB host controller extension driver (*UcxVersion.sys*) provides a layer of abstraction between the USB 3.0 hub driver and the USB 3.0 host controller driver. The extension driver has its own representation of host controllers, devices, and endpoints. The outputs of the commands in the [!lucx_controller_list](#) family are based on the data structures maintained by the extension driver. For more information about the USB host controller extension driver and the USB 3.0 host controller driver, see [USB Driver Stack Architecture](#). For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the [USB Debugging Innovations in Windows 8](#) video.

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.usbanalyze

Article • 10/25/2023

The [!usb3kd.usbanalyze](#) extension analyzes a USB 3.0 bug check.

dbgcmd

```
!usb3kd.usbanalyze [-v]
```

Parameters

-v

The display is verbose.

DLL

Usb3kd.dll

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

[BUGCODE_USB3_DRIVER](#)

!usb3kd.usbdstatus

Article • 10/25/2023

The [!usb3kd.usbdstatus](#) extension displays the name of a USBD status code.

```
dbgcmd
```

```
!usb3kd.uxc_usbdstatus UrbStatus
```

Parameters

UsbdStatus

The numeric value of a USBD status code.

DLL

Usb3kd.dll

Remarks

USBD status codes are defined in Usb.h.

Examples

The following example passes the numeric value 0x80000200 to the [!usbdstatus](#) command. The command returns the name of the status code, USBD_STATUS_INVALID_URB_FUNCTION.

```
dbgcmd
```

```
3: kd> !usbdstatus 0x80000200
USBD_STATUS_INVALID_URB_FUNCTION (0x80000200)
```

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.usb_tree

Article • 10/25/2023

The [!usb3kd.usb_tree](#) extension displays information, in tree format, about all USB 3.0 controllers, hubs, and devices on the computer.

```
dbgcmd
!usb3kd.usb_tree [1]
```

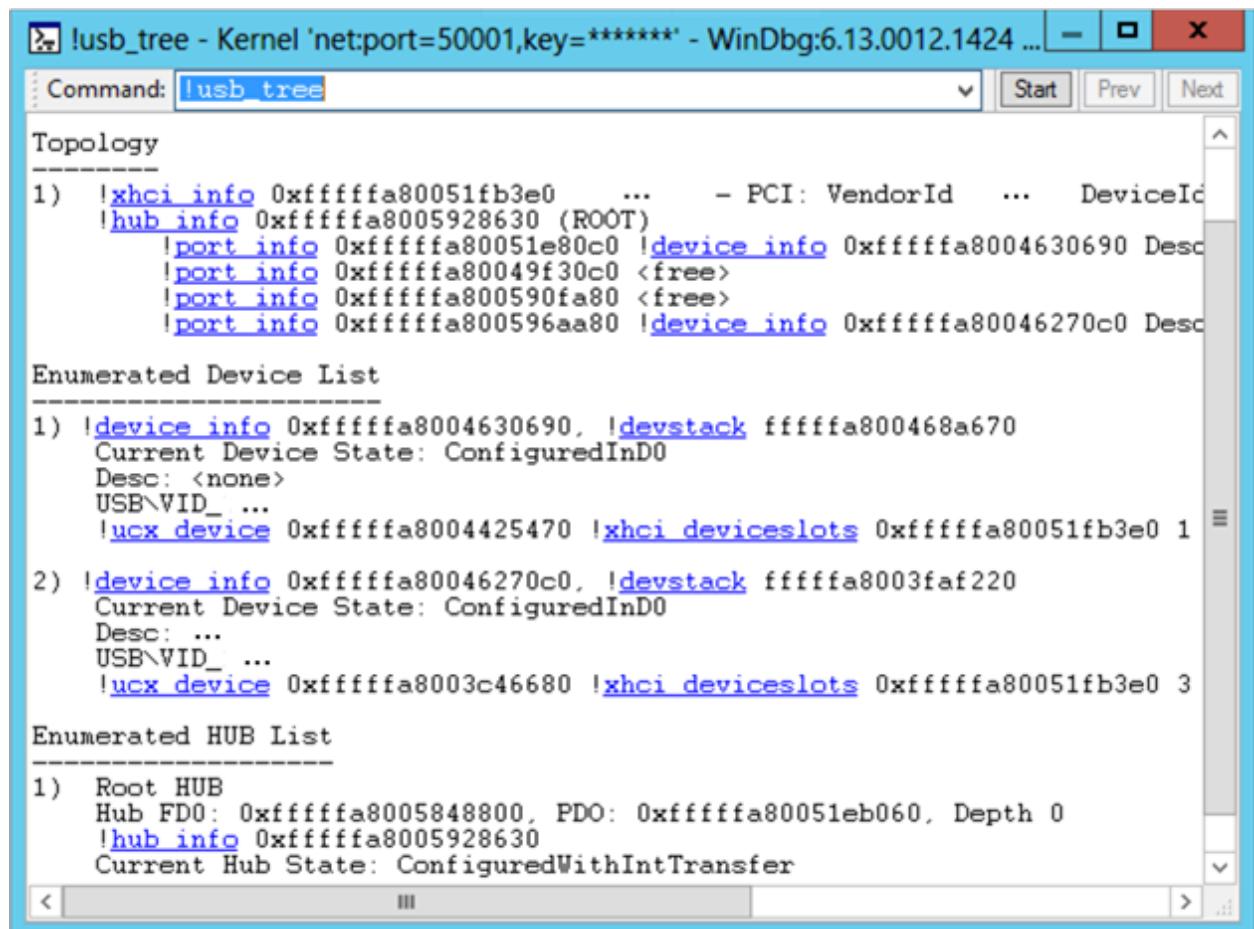
Parameters

1

The display includes status information for hubs and ports.

Examples

The following screen shot shows the output of the [!usb_tree](#) command.



The screenshot shows the WinDbg debugger interface with the command `!usb_tree` entered in the command window. The output is displayed in three sections: Topology, Enumerated Device List, and Enumerated HUB List.

Topology

```
1) !xhci_info 0xfffffa80051fb3e0 ... - PCI: VendorId ... DeviceId
   !hub_info 0xfffffa8005928630 (ROOT)
     !port_info 0xfffffa80051e80c0 !device_info 0xfffffa8004630690 Desc
     !port_info 0xfffffa80049f30c0 <free>
     !port_info 0xfffffa800590fa80 <free>
     !port_info 0xfffffa800596aa80 !device_info 0xfffffa80046270c0 Desc
```

Enumerated Device List

```
1) !device_info 0xfffffa8004630690, !devstack fffffa800468a670
   Current Device State: ConfiguredInD0
   Desc: <none>
   USB\VID_ ...
   !ucx_device 0xfffffa8004425470 !xhci_deviceslots 0xfffffa80051fb3e0 1
```

```
2) !device_info 0xfffffa80046270c0, !devstack fffffa8003faf220
   Current Device State: ConfiguredInD0
   Desc: ...
   USB\VID_ ...
   !ucx_device 0xfffffa8003c46680 !xhci_deviceslots 0xfffffa80051fb3e0 3
```

Enumerated HUB List

```
1) Root HUB
   Hub FDO: 0xfffffa8005848800, PDO: 0xfffffa80051eb060, Depth 0
   !hub_info 0xfffffa8005928630
   Current Hub State: ConfiguredWithIntTransfer
```

The output shows that there is one USB 3.0 host controller, which is represented by the line that begins with [!xhci_info](#). The next line represents the root hub for the host controller. The next four lines represent ports associated with the root hub. You can see that two ports have devices connected.

The output uses [Using Debugger Markup Language \(DML\)](#) to provide links. The links execute commands that give detailed information about individual objects in the tree. For example, you could get information about one of the connected devices by clicking one of the [!device_info](#) links. As an alternative to clicking a link, you can enter a command. For example, to see information about the first connected device, you could enter the command `!device_info 0xfffffa8004630690`.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The [!usb_tree](#) command is the parent command for this set of commands.

- [!hub_info](#)
- [!hub_info_from_fdo](#)
- [!device_info](#)
- [!device_info_from_pdo](#)
- [!port_info](#)

The information displayed by the [!usb_tree](#) family of commands is based on data structures maintained by the USB 3.0 hub driver. For information about the USB 3.0 hub driver and other drivers in the USB 3.0 stack, see [USB Driver Stack Architecture](#). For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the [USB Debugging Innovations in Windows 8](#) video.

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.urb

Article • 10/25/2023

The **!usb3kd.urb** extension displays information about a USB request block (URB).

```
dbgcmd
```

```
!usb3kd.urb UrbAddress
```

Parameters

UrbAddress

Address of the URB.

DLL

Usb3kd.dll

Examples

The following example shows the address of a URB (0xfffffa8005a2cbe8) in the output of the **!xhci_deviceslots** command.

```
dbgcmd
```

```
3: kd> !xhci_deviceslots 0xfffffa800520d2d0
## Dumping dt _DEVICESLOT_DATA 0xfffffa8003612e80
-----
DeviceContextBase: VA 0xfffffa8005a64000 LA 0x116864000 !wdfcommonbuffer
0x57ffa7ca758 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa80059027d0 dt
_SLOT_CONTEXT32 0xfffffa8005a65000
-----
-----
USB\VID_...
SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
...
PendingTransferList:
[0] dt _TRANSFER_DATA 0xfffffa80059727f0 !urb
0xfffffa8005a2cbe8 !wdfrequest 0x57ffa68d998 TransferState_Pending
...
```

The following example passes the address of the URB to the !usb3kd.urb command.

```
dbgcmd
```

```
3: kd> !urb 0xfffffa8005a2cbe8

## Dumping URB 0xfffffa8005a2cbe8
-----
Function:          URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER (0x9)
UsbdDeviceHandle:
    !ucx_device 0xfffffa8005901840
    !xhci_deviceslots 0xfffffa800520d2d0 1

Status:           USBD_STATUS_PENDING (0x40000000)
UsbdFlags:        (0x0)
dt _URB_BULK_OR_INTERRUPT_TRANSFER 0xfffffa8005a2cbe8
PipeHandle:       0xfffffa800596f720
TransferFlags:    (0x1) USBD_TRANSFER_DIRECTION_IN
TransferBufferLength: 0x0
TransferBuffer:   0xfffffa8005a2cc88
TransferBufferMDL: 0xfffffa8005848930
```

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_capability

Article • 10/25/2023

The [!usb3kd.xhci_capability](#) extension displays the capabilities of a USB 3.0 host controller.

```
dbgcmd
```

```
!usb3kd.xhci_capability DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the [!xhci_capability](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB Driver Stack Architecture](#).

Examples

To obtain the address of the device extension, look at the output of the [!xhci_dumpall](#) command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
dbgcmd
```

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-----
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...
```

```
dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
...
```

Now you can pass the address of the device extension to the !xhci_capability command.

```
dbgcmd

3: kd> !xhci_capability 0xfffffa800536e2d0

## Controller Capabilities
-----
dt USBXHCI!_REGISTER_DATA 0xfffffa8005362c00
dt USBXHCI!_CAPABILITY_REGISTERS 0xfffff880046a8000
MajorRevision.MinorRevision = 0.96
Device Slots: 32
Interrupters: 8
Ports: 4
IsochSchedulingThreshold: 1
EventRingSegmentTableMax: 1 (2^ERST = 2)
ScratchpadRestore: OFF
MaxScratchpadBuffers: 0
U1DeviceExitLatency: 0
U2DeviceExitLatency: 0
AddressingCapability: 64 bit
BwNegotiationCapability: ON
ContextSize: 32 bytes
PortPowerControl: ON
PortIndicators: OFF
LightHCResetCapability: OFF
LatencyToleranceMessagingCapability: ON
NoSecondarySidSupport: TRUE
MaximumPrimaryStreamArraySize = 4 ( 2^(MaxPSASize+1) = 32 )
XhciExtendedCapabilities:
[1] USB_LEGACY_SUPPORT: dt _USBLEGSUP 0xfffff880046a8500
[2] Supported Protocol 0xfffff880046a8510, Version 3.0, Offset 1,
Count 2, HighSpeedOnly OFF, IntegratedHub OFF, HardwareLPM OFF
[3] Supported Protocol 0xfffff880046a8520, Version 2.0, Offset 3,
Count 2, HighSpeedOnly OFF, IntegratedHub OFF, HardwareLPM OFF

## Software Supported Capabilities
-----
DeviceSlots: 32
Interrupters: 1
Ports: 4
MaxEventRingSegments: 2
U1DeviceExitLatency: 0
U2DeviceExitLatency: 0
DeviceFlags:
IgnoreBiosHandoffFailure
```

```
SetLinkTrbChainBit  
UseSingleInterrupter  
DisableIdlePowerManagement
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_commandring

Article • 10/25/2023

The [!usb3kd.xhci_commandring](#) extension displays information about the command ring data structure associated with a USB 3.0 host controller.

```
dbgcmd
```

```
!usb3kd.xhci_commandring DeviceExtension
```

Parameters

DeviceExtension

AAddress of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the [!xhci_commandring](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB Driver Stack Architecture](#).

The command ring is a data structure used by the USB 3.0 host controller driver to pass commands to the host controller.

Examples

To obtain the address of the device extension, look at the output of the [!xhci_dumpall](#) command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
dbgcmd
```

```
3: kd> !xhci_dumpall
```

```
## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-----
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
!xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
...

```

Now you can pass the address of the device extension to the **!xhci_commandring** command.

```
dbgcmd

3: kd> !xhci_commandring 0xfffffa800536e2d0

## Dumping dt _COMMAND_DATA 0xfffffa8005362f70 !rcdrlogdump USBXHCI -a
0xfffffa8005a8f010
-----
-----
Stop: OFF Abort: OFF Running: ON
CommandRingBufferData: VA 0xfffffa8005aeb200 LA 0x1168eb200 !wdfcommonbuffer
0x57ffa65d988 Size 512
DequeueIndex: 24 EnqueueIndex: 24 CycleState: 0

Command Ring TRBs:
[ 0] Unknown TRB Type 49 0xfffffa8005aeb200

[ 1] ENABLE_SLOT           0xfffffa8005aeb210 CycleBit 1
[ 2] ADDRESS_DEVICE        0xfffffa8005aeb220 CycleBit 1
SlotId 1 BlockSetAddressRequest 1
...
PendingList:
Empty List

WaitingList:
Empty List
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_deviceslots

Article • 10/25/2023

The **!usb3kd.xhci_deviceslots** extension displays information about the devices connected to a USB 3.0 host controller.

dbgcmd

```
!usb3kd.xhci_deviceslots DeviceExtension [SlotNumber] [verbose]
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

SlotNumber

Slot number of the device to be displayed. If this parameter is omitted, all devices are displayed.

verbose

The display is verbose.

DLL

Usb3kd.dll

Remarks

The output the **!xhci_deviceslots** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB Driver Stack Architecture](#).

The USB 3.0 host controller driver maintains a list of data structures that represent the devices connected to the controller. Each of these data structures is identified by a slot number.

Examples

To obtain the address of the device extension, look at the output of the `!xhci_dumpall` command. In the following example, the address of the device extension is `0xfffffa800536e2d0`.

```
dbgcmd

3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-----
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...

dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
!xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
!xhci_deviceslots 0xfffffa800536e2d0
...
...
```

Now you can pass the address of the device extension to the `!usb3kd.xhci_deviceslots` command.

```
dbgcmd

3: kd> !xhci_deviceslots 0xfffffa800536e2d0

## Dumping dt _DEVICESLOT_DATA 0xfffffa8005226220
-----
DeviceContextBase: VA 0xfffffa8005ab9000 LA 0x1168b9000 !wdfcommonbuffer
0x57ffa65c9b8 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa8005a427d0 dt
_SLOT_CONTEXT32 0xfffffa8005aba000
-----
-----
USB\VID_125F&PID_312A ADATA Technology Co., Ltd.
SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
Speed: Super PortPathDepth: 1 PortPath: [ 2 ] DeviceAddress: 1
!device_info_from_pdo 0xfffffa8005a36800
DeviceContextBuffer: VA 0xfffffa8005aba000 LA 0x1168ba000
!wdfcommonbuffer 0x57ffa656948 Size 4096
InputDeviceContextBuffer: VA 0xfffffa8005b65000 LA 0x116965000
!wdfcommonbuffer 0x57ffa5be958 Size 4096

[1] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0xfffffa8005a126f0 dt
_ENDPOINT_CONTEXT32 0xfffffa8005aba020 ES_RUNNING
-----
```

```

EndpointType_Control Address: 0x0 PacketSize: 512 Interval: 0
!ucx_endpoint 0xfffffa8005a3f710
RecorderLog: !rcdrlogdump USBXHCI -a 0xfffffa8005b60010

[0] dt _TRANSFERRING_DATA 0xfffffa8005b64ec0 Events: 0x0
TransferRingState_Idle
...

## [2] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa80052de320 dt
_SLOT_CONTEXT32 0xfffffa8005b8b000
-----
-----

USB\VID_18A5&PID_0304 Verbatim Americas LLC
SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
Speed: High PortPathDepth: 1 PortPath: [ 3 ] DeviceAddress: 2
!device_info_from_pdo 0xfffffa80058fb800
DeviceContextBuffer: VA 0xfffffa8005b8b000 LA 0x11698b000
!wdfcommonbuffer 0x57ffa426b18 Size 4096
InputDeviceContextBuffer: VA 0xfffffa8005b8c000 LA 0x11698c000
!wdfcommonbuffer 0x57ffadbe3c8 Size 4096

[1] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0xfffffa800714b050 dt
_ENDPOINT_CONTEXT32 0xfffffa8005b8b020 ES_RUNNING
-----
-----

EndpointType_Control Address: 0x0 PacketSize: 64 Interval: 0
!ucx_endpoint 0xfffffa80036a20c0
RecorderLog: !rcdrlogdump USBXHCI -a 0xfffffa8005bd0b60

[0] dt _TRANSFERRING_DATA 0xfffffa8004ed8df0 Events: 0x0
TransferRingState_Idle
-----

...

```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_dumpall

Article • 10/25/2023

The [!usb3kd.xhci_dumpall](#) command displays information about all USB 3.0 host controllers on the computer. The display is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys).

```
dbgcmd
!usb3kd.xhci_dumpall [1]
```

Parameters

1

Runs all of the XHCI commands and displays the output of each command.

Examples

The following screen shot show the output of the [!xhci_dumpall](#) command.

```
!xhci_dumpall - Kernel 'net:port=50001,key=*****' - WinDbg:6.13.0012.142...
Command: !xhci_dumpall
Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware
dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
!xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
!xhci_deviceslots 0xfffffa800536e2d0
!xhci_eventring 0xfffffa800536e2d0
!xhci_resourceusage 0xfffffa800536e2d0
!pci 100 0x30 0x0 0x0
```

The output shows that there is one USB 3.0 host controller.

The output uses [Using Debugger Markup Language \(DML\)](#) to provide links. The links execute commands that give detailed information about the state of the host controller as it is maintained by the USB 3.0 host controller driver. For example, you could get detailed information about the host controller capabilities by clicking the [!xhci_capability](#) link. As an alternative to clicking a link, you can enter a command. For

example, to see information about the host controller's resource usage, you could enter the command `!xhci_resourceusage 0xfffffa800536e2d0`.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The `!xhci_dumpall` command is the parent command for this set of commands.

- [!xhci_capability](#)
- [!xhci_info](#)
- [!xhci_deviceslots](#)
- [!xhci_commandring](#)
- [!xhci_eventring](#)
- [!xhci_transferring](#)
- [!xhci_trb](#)
- [!xhci_registers](#)
- [!xhci_resourceusage](#)

The information displayed by the `!xhci_dumpall` family of commands is based on data structures maintained by the USB 3.0 host controller driver. For information about the USB 3.0 host controller driver and other drivers in the USB 3.0 stack, see [USB Driver Stack Architecture](#). For an explanation of the data structures used by the drivers in the USB 3.0 stack, see Part 2 of the [USB Debugging Innovations in Windows 8](#) video.

See also

[USB 3.0 Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_eventring

Article • 10/25/2023

The [!usb3kd.xhci_eventring](#) extension displays information about the event ring data structure associated with a USB 3.0 host controller.

dbgcmd

```
!usb3kd.xhci_eventring DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output [!xhci_eventring](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

The event ring is a structure used by the USB 3.0 host controller to inform drivers that an action has completed.

Examples

To obtain the address of the device extension, look at the output of the [!xhci_dumpall](#) command. In the following example, the address of the device extension is 0xfffffa80053072f0.

dbgcmd

```
3: kd> !xhci_dumpall
```

```
## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
```

```

-----  

1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...  

    dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0  

    !rcdrlogdump USBXHCI -a 0xfffffa8005068520  

    !rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)  

    !wdfdevice 0x57ffac91fd8  

    !xhci_capability 0xfffffa800536e2d0  

    !xhci_registers 0xfffffa800536e2d0  

    !xhci_commandring 0xfffffa800536e2d0 (No commands are pending)  

    !xhci_deviceslots 0xfffffa800536e2d0  

    !xhci_eventring 0xfffffa800536e2d0  

    ...

```

Now you can pass the address of the device extension to the `!xhci_eventring` command.

dbgcmd

```

3: kd> !xhci_eventring 0xfffffa800536e2d0  

## Dumping dt _PRIMARY_INTERRUPTER_DATA ffffffa800536b5b0
-----  

## [0] Interrupter : dt _INTERRUPTER_DATA 0xfffffa800536b7d0 !rcdrlogdump
USBXHCI -a 0xfffffa8005aeab60
-----  

-----  

    DequeueSegment: 1 DequeueIndex: 217 TotalEventRingSegments: 2  

    TRBsPerSegment: 256  

    CurrentBufferData : VA 0xfffffa8005373000 LA 0x117173000  

    !wdfcommonbuffer 0x57ffa65b9b8 Size 4096  

    EventRingTableBufferData : VA 0xfffffa8005aeb000 LA 0x1168eb000  

    !wdfcommonbuffer 0x57ffa65d988 Size 512  

    [0] VA 0xfffffa8005370000 LA 0x117170000 !wdfcommonbuffer 0x57ffa6599b8  

    Size 4096  

    [1] VA 0xfffffa8005373000 LA 0x117173000 !wdfcommonbuffer 0x57ffa65b9b8  

    Size 4096  

    Event Ring TRBs:  

    [207] TRANSFER_EVENT      0xfffffa8005373cf0 CycleBit 0 SlotId  2  

    EndpointID 4 EventData 1 Pointer 0xfffffa8005366700 CC_SUCCESS  

    [208] TRANSFER_EVENT      0xfffffa8005373d00 CycleBit 0 SlotId  2  

    EndpointID 3 EventData 1 Pointer 0xfffffa8005a3d850 CC_SHORT_PACKET  

    [209] TRANSFER_EVENT      0xfffffa8005373d10 CycleBit 0 SlotId  1  

    EndpointID 4 EventData 1 Pointer 0xfffffa8005a3d850 CC_SUCCESS  

    [210] TRANSFER_EVENT      0xfffffa8005373d20 CycleBit 0 SlotId  1  

    EndpointID 3 EventData 1 Pointer 0xfffffa8005366700 CC_SUCCESS  

    [211] TRANSFER_EVENT      0xfffffa8005373d30 CycleBit 0 SlotId  2  

    EndpointID 4 EventData 1 Pointer 0xfffffa8005366700 CC_SUCCESS  

    [212] TRANSFER_EVENT      0xfffffa8005373d40 CycleBit 0 SlotId  2  

    EndpointID 3 EventData 1 Pointer 0xfffffa8005a3d850 CC_SHORT_PACKET  

    [213] TRANSFER_EVENT      0xfffffa8005373d50 CycleBit 0 SlotId  1

```

```
EndpointID 4 EventData 1 Pointer 0xfffffa8005a3d850 CC_SUCCESS
    [214] TRANSFER_EVENT      0xfffffa8005373d60 CycleBit 0 SlotId 1
EndpointID 3 EventData 1 Pointer 0xfffffa8005366700 CC_SUCCESS
    [215] TRANSFER_EVENT      0xfffffa8005373d70 CycleBit 0 SlotId 2
EndpointID 4 EventData 1 Pointer 0xfffffa8005366700 CC_SUCCESS
    [216] TRANSFER_EVENT      0xfffffa8005373d80 CycleBit 0 SlotId 2
EndpointID 3 EventData 1 Pointer 0xfffffa8005a3d850 CC_SHORT_PACKET
```

See also

[USB 3.0 Extensions](#)

[**!xhci_dumpall**](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_findowner

Article • 10/25/2023

The **!usb3kd.xhci_findowner** command finds the owner a common buffer.

```
dbgcmd
```

```
!usb3kd.xhci_findowner Address
```

Parameters

Address

Virtual or physical address of a common buffer.

DLL

Usb3kd.dll

Remarks

A common buffer is a block of physically contiguous memory that's addressable by hardware. The USB 3.0 driver stack uses common buffers to communicate with USB 3.0 host controllers. Suppose the system crashes, and you come across an address that you suspect might be common buffer memory. If the address is common buffer memory, this command tells you which USB 3.0 host controller the memory belongs to (in case you have more than one USB 3.0 controller) and what the memory is used for.

Examples

The following example calls [!xhci_resourceusage](#) to list the addresses of some common buffers.

```
dbgcmd
```

```
0: kd> !usb3kd.xhci_resourceusage 0x867fbff0

## Dumping CommonBuffer Resources
-----
dt USBXHCI!_COMMON_BUFFER_DATA 0x868d61c0
DmaEnabler:!wdfdmaenabler 0x79729fe8
```

```

CommonBuffers Large: Total 8 Available 2 Used 6 TotalBytes 32768
    [ 1] dt _TRACKING_DATA 0x868d73a4 VA 0x868e0000 LA 0xdb2e0000 --
Owner 0x86801690 Tag: Int2 Size 4096
    [ 2] dt _TRACKING_DATA 0x868d6d1c VA 0x868e1000 LA 0xdb2e1000 --
Owner 0x86801690 Tag: Int2 Size 4096
    [ 3] dt _TRACKING_DATA 0x868d6c54 VA 0x868e2000 LA 0xdb2e2000 --
Owner 0x86801690 Tag: Int2 Size 4096
    [ 4] dt _TRACKING_DATA 0x868d6b8c VA 0x868e3000 LA 0xdb2e3000 --
Owner 0x86801690 Tag: Int2 Size 4096
    [ 5] dt _TRACKING_DATA 0x868d67b4 VA 0x868e5000 LA 0xdb2e5000 --
Owner 0x86801548 Tag: Slt1 Size 4096
    [ 6] dt _TRACKING_DATA 0x868d50b4 VA 0x868e6000 LA 0xdb2e6000 --
Owner 0x86801548 Tag: Slt3 Size 4096

CommonBuffers Small: Total 16 Available 13 Used 3 TotalBytes 8192
    [ 1] dt _TRACKING_DATA 0x868d6974 VA 0x868e4000 LA 0xdb2e4000 --
Owner 0x86801690 Tag: Int1 Size 512
    [ 2] dt _TRACKING_DATA 0x868d69a4 VA 0x868e4200 LA 0xdb2e4200 --
Owner 0x86801548 Tag: Slt2 Size 512
    [ 3] dt _TRACKING_DATA 0x868d69d4 VA 0x868e4400 LA 0xdb2e4400 --
Owner 0x86801488 Tag: Cmd1 Size 512

```

One of the virtual addresses listed in the preceding output is 0x868e2000. The following example passes that address to `!xhci_findowner`. One of the physical addresses listed in the preceding output is 0xdb2e4400. The following example passes 0xdb2e4440 (offset 0x40 bytes from 0xdb2e4400) to `!xhci_findowner`.

```

dbgcmd

0: kd> !xhci_findowner 0x868e2000

!xhci_info 0x867fbff0 Texas Instruments - PCI: VendorId 0x104c DeviceId
0x8241 RevisionId 0x02

    dt _TRACKING_DATA 0x868d6c54 VA 0x868e2000 LA 0xdb2e2000 -- Owner
0x86801690 Tag: Int2 Size 4096

    dt _INTERRUPTER_DATA 0x86801690
    !xhci_eventring 0x867fbff0 <-- This memory is used for event ring.

0: kd> !xhci_findowner 0xdb2e4440 <-- Note the offset difference.

!xhci_info 0x867fbff0 Texas Instruments - PCI: VendorId 0x104c DeviceId
0x8241 RevisionId 0x02

    dt _TRACKING_DATA 0x868d69d4 VA 0x868e4400 LA 0xdb2e4400 -- Owner
0x86801488 Tag: Cmd1 Size 512

```

```
dt _COMMAND_DATA 0x86801488
!xhci_commandring 0x867fbff0 <-- This memory is used for command ring.
```

The `!xhci_findowner` command is especially useful when you have an address in a transfer request block (TRB), and you want to track it back to the device slot that it belongs to. In the following example, one of the addresses listed in the output of `!xhci_transferring` is 0xda452230, which is the physical address of a TRB. The example passes that address to `!xhci_findowner`. The output shows that the TRB belongs to device slot 8 (`!xhci_deviceslots 0x8551d370 8`).

```
dbgcmd

0: kd> !usb3kd.xhci_transferring 0x87652200

    [ 0] NORMAL      0xda452200 CycleBit 1 IOC 0 BEI 0
InterrupterTarget 2 TransferLength      6 TDSize  0
    [ 1] EVENT_DATA  0xda452210 CycleBit 1 IOC 1 BEI 0
InterrupterTarget 2 Data 0x8511375c TotalBytes 6
    [ 2] NORMAL      0xda452220 CycleBit 1 IOC 0 BEI 0
InterrupterTarget 2 TransferLength      6 TDSize  0
    [ 3] EVENT_DATA  0xda452230 CycleBit 1 IOC 1 BEI 0
InterrupterTarget 2 Data 0x857d076c TotalBytes 6

0: kd> !xhci_findowner 0xda452230

!xhci_info 0x8551d370 Renesas - PCI: VendorId 0x1912 DeviceId 0x0015
RevisionId 0x02 Firmware 0x0020.0006

    dt _TRACKING_DATA 0x8585fd5c VA 0x87652200 LA 0xda452200 -- Owner
0x85894548 Tag: Rng1 Size 512

    !xhci_deviceslots 0x8551d370 8

    [0] dt _TRANSFERRING_DATA 0x85894548 Events: 0x0 TransferRingState_Idle
-----
-----
    WdfQueue: !wdfqueue 0x7a76bcb0 (0 waiting)
    CurrentRingBufferData: VA 0x87652200 LA 0xda452200 !wdfcommonbuffer
0x7a7a0370 Size 512
    Current: !xhci_transferring 0x87652200
    PendingTransferList:
        [0] dt _TRANSFER_DATA 0x851136f0 !urb 0x84e55468 !wdfrequest
0x7aeec9e8 TransferState_Pending
        [1] dt _TRANSFER_DATA 0x857d0700 !urb 0x85733be8 !wdfrequest
0x7a82f9d8 TransferState_Pending
```

See also

[USB 3.0 Extensions](#)

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_info

Article • 10/25/2023

The **!usb3kd.xhci_info** extension displays all the XHCI commands for an individual USB 3.0 host controller.

```
dbgcmd
```

```
!usb3kd.xhci_info DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the **!usb3kd.xhci_info** command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

Examples

You can get address of the device extension from the **!xhci_dumpall** command or from a variety of other debugger commands. For example, the **!devstack** command displays the address of the device extension. In the following example, the address of the device extension for the host controller's FDO is ffffffa800536e2d0.

```
dbgcmd
```

```
3: kd> !devnode 0 1 usbxhci
Dumping TopRootDeviceNode (= 0xfffffa8003609cc0)
DevNode 0xfffffa8003df3010 for PDO 0xfffffa8003dd5870
  InstancePath is "PCI\VEN_...
  ServiceName is "USBXHCI"
```

```
...
3: kd> !devstack 0xfffffa8003dd5870
!DevObj   !DrvObj          !DevExt   ObjectName
fffffa800534b060  \Driver\USBXHCI    ffffffa800536e2d0  USBFDO-3
fffffa8003db5790  \Driver\ACPI      ffffffa8003701cb0
> ffffffa8003dd5870  \Driver\pci      ffffffa8003dd59c0  NTPNP_PCI0020
...
```

Now you can pass the address of the device extension to the `!xhci_info` command.

```
dbgcmd

3: kd> !xhci_info 0xfffffa80`0536e2d0
## Dumping XHCI controller commands - DeviceExtension 0xfffffa800536e2d0
-----
... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...
dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
!xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
!xhci_deviceslots 0xfffffa800536e2d0
!xhci_eventring 0xfffffa800536e2d0
!xhci_resourceusage 0xfffffa800536e2d0
!pci 100 0x30 0x0 0x0
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_registers

Article • 10/25/2023

The [!usb3kd.xhci_registers](#) extension displays the registers of a USB 3.0 host controller.

```
dbgcmd
```

```
!usb3kd.xhci_registers DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the [!xhci_registers](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

Examples

To obtain the address of the device extension, look at the output of the [!xhci_dumpall](#) command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
dbgcmd
```

```
3: kd> !xhci_dumpall
## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-----
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...
dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
```

```
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
...
```

Now you can pass the address of the device extension to the `!xhci_registers` command.

```
dbgcmd

3: kd> !xhci_registers 0xfffffa800536e2d0
## Dumping controller registers
-----
dt USBXHCI!_OPERATIONAL_REGISTERS 0xfffff880046a8020
DeviceContextBaseAddressArrayPointer: 00000001168b9000

Command Registers
-----
RunStopBit: 1
HostControllerReset: 0
...
Status Registers
-----
HcHalted: 0
HostSystemError: 0
...
commandRingControl Registers
-----
RingCycleState: 0
CommandStop: 0
...
Runtime Registers
-----
dt USBXHCI!_RUNTIME_REGISTERS 0xfffff880046a8600
MicroFrameIndex: 0x3f7a

dt -ba8 USBXHCI!_INTERRUPTER_REGISTER_SET 0xfffff880046a8620

RootPort Registers
-----
dt -a4 -r2 USBXHCI!_PORT_REGISTER_SET 0xfffff880046a8420
```

See also

[USB 3.0 Extensions](#)

!xhci_dumpall

Universal Serial Bus (USB) Drivers

!usb3kd.xhci_resourceusage

Article • 10/25/2023

The [!usb3kd.xhci_resourceusage](#) extension displays the resources used by a USB 3.0 host controller.

```
dbgcmd
```

```
!usb3kd.xhci_resourceusage DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the host controller's functional device object (FDO).

DLL

Usb3kd.dll

Remarks

The output the [!xhci_resourceusage](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

Examples

To obtain the address of the device extension, look at the output of the [!xhci_dumpall](#) command. In the following example, the address of the device extension is 0xfffffa800536e2d0.

```
dbgcmd
```

```
3: kd> !xhci_dumpall

## Dumping all the XHCI controllers - DrvObj 0xfffffa80053072f0
-----
1) ... - PCI: VendorId ... DeviceId ... RevisionId ... Firmware ...
```

```
dt USBXHCI!_CONTROLLER_DATA 0xfffffa80052f20c0
!rcdrlogdump USBXHCI -a 0xfffffa8005068520
!rcdrlogdump USBXHCI -a 0xfffffa8004e8b9a0 (rundown)
!wdfdevice 0x57ffac91fd8
!xhci_capability 0xfffffa800536e2d0
!xhci_registers 0xfffffa800536e2d0
!xhci_commandring 0xfffffa800536e2d0 (No commands are pending)
!xhci_deviceslots 0xfffffa800536e2d0
!xhci_eventring 0xfffffa800536e2d0
!xhci_resourceusage 0xfffffa800536e2d0
...
...
```

Now you can pass the address of the device extension to the **!xhci_resourceusage** command.

dbgcmd

```
3: kd> !xhci_resourceusage 0xfffffa800536e2d0

## Dumping CommonBuffer Resources
-----
dt USBXHCI!_COMMON_BUFFER_DATA 0xfffffa80059a5920
DmaEnabler:!wdfdmaenabler 0x57ffa65a9c8

CommonBuffers Large: Total 9 Available 2 Used 7 TotalBytes 36864
    [ 1] dt _TRACKING_DATA 0xfffffa80059a6768 VA 0xfffffa8005370000 LA
0x117170000 ...
    [ 2] dt _TRACKING_DATA 0xfffffa80059a4768 VA 0xfffffa8005373000 LA
0x117173000 ...
    ...
CommonBuffers Small: Total 32 Available 8 Used 24 TotalBytes 16384
    [ 1] dt _TRACKING_DATA 0xfffffa80059a2798 VA 0xfffffa8005aeb000 LA
0x1168eb000 ...
    [ 2] dt _TRACKING_DATA 0xfffffa80059a27e8 VA 0xfffffa8005aeb200 LA
0x1168eb200 ...
    ...
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_trb

Article • 10/25/2023

The [!usb3kd.xhci_trb](#) extension displays one or more transfer request blocks (TRBs) used by a USB 3.0 host controller

dbgcmd

```
!usb3kd.xhci_trb VirtualAddress Count  
!usb3kd.xhci_trb PhysicalAddress Count 1
```

Parameters

VirtualAddress

Virtual address of a TRB.

PhysicalAddress

Physical address of a TRB.

Count

The number of consecutive TRBs to display, starting at *VirtualAddress* or *PhysicalAddress*.

1

Specifies that the address is a physical address.

DLL

Usb3kd.dll

Remarks

The output the [!xhci_trb](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

Examples

In the following example, **0x844d7c00** is the virtual address of a TRB. The **1** is the count, which specifies how many consecutive TRBs to display.

```
dbgcmd
```

```
0: kd> !xhci_trb 0x844d7c00 1

[ 0] ISOCH      0x844d7c00 CycleBit 1 IOC 0 CH 1 BEI 0
InterrupterTarget 1 TransferLength 2688 TDSIZE 0 TBC 0 TLBPC 2 Frame 0x3D2
```

In the following example, **0x0dc0d7c00** is the physical address of a TRB. The **4** is the count, which specifies how many consecutive TRBs to display. The **1** specifies that the address is a physical address.

```
dbgcmd
```

```
0: kd> !xhci_trb 0x0dc0d7c00 4 1

[ 0] ISOCH      0xdced7c00 CycleBit 1 IOC 0 CH 1 BEI 0
InterrupterTarget 1 TransferLength 2688 TDSIZE 0 TBC 0 TLBPC 2 Frame 0x3D2
[ 1] EVENT_DATA  0xdced7c10 CycleBit 1 IOC 1 CH 0 BEI 1
InterrupterTarget 1 Data 0x194c9bcf001b0001 PacketId 27 Frame 0x194c9bcf
TotalBytes 2688
[ 2] ISOCH      0xdced7c20 CycleBit 1 IOC 0 CH 1 BEI 0
InterrupterTarget 1 TransferLength 1352 TDSIZE 2 TBC 0 TLBPC 2 Frame 0x3D2
[ 3] NORMAL     0xdced7c30 CycleBit 1 IOC 0 CH 1 BEI 0
InterrupterTarget 1 TransferLength 1336 TDSIZE 0
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usb3kd.xhci_transferring

Article • 10/25/2023

The [!usb3kd.xhci_transferring](#) extension displays a transfer ring (used by a USB 3.0 host controller) until it detects a cycle bit change.

dbgcmd

```
!usb3kd.xhci_transferring VirtualAddress  
!usb3kd.xhci_transferring PhysicalAddress 1
```

Parameters

VirtualAddress

Virtual address of the transfer ring.

PhysicalAddress

Physical address of the transfer ring.

1

Specifies that the address is a physical address.

DLL

Usb3kd.dll

Remarks

The output of the [!xhci_transferring](#) command is based on the data structures maintained by the USB 3.0 host controller driver (UsbXhci.sys). For more information about the USB 3.0 host controller driver and other drivers in the USB stack, see [USB host-side drivers in Windows](#).

The transfer ring is a structure used by the USB 3.0 host controller driver to maintain a list of transfer request blocks (TRBs). This command takes the virtual or physical address of a transfer ring, but displays the physical address of the TRBs. This is done so the command can correctly traverse LINK TRBs.

Examples

To obtain the address of the transfer ring, look at the output of the [!xhci_deviceslots](#) command. In the following example, the virtual address of the transfer ring is 0xfffffa8005b2fe00.

```
dbgcmd

3: kd> !usb3kd.xhci_deviceslots 0xfffffa800523a2d0

## Dumping dt _DEVICESLOT_DATA 0xfffffa80051a3300
-----
DeviceContextBase: VA 0xfffffa8005a41000 LA 0x116841000 !wdfcommonbuffer
0x57ffa6ff9b8 Size 4096

## [1] SlotID : dt USBXHCI!_USBDEVICE_DATA 0xfffffa800598c7d0 dt
_SLOT_CONTEXT32 0xfffffa8005a42000
-----
-----
USB\VID_125F&PID_312A ADATA Technology Co., Ltd.
SlotEnabled IsDevice NumberOfTTs 0 TTThinkTime 0
Speed: Super PortPathDepth: 1 PortPath: [ 2 ] DeviceAddress: 1
!device_info_from_pdo 0xfffffa800597d720
DeviceContextBuffer: VA 0xfffffa8005a42000 LA 0x116842000
!wdfcommonbuffer 0x57ffa7009b8 Size 4096
InputDeviceContextBuffer: VA 0xfffffa8005b2d000 LA 0x11692d000
!wdfcommonbuffer 0x57ffa674958 Size 4096
...
[3] DeviceContextIndex : dt USBXHCI!_ENDPOINT_DATA 0xfffffa8005b394e0 dt
_ENDPOINT_CONTEXT32 0xfffffa8005a42060 ES_RUNNING
-----
-----
...
CurrentRingBufferData: VA 0xfffffa8005b2fe00 LA 0x11692fe00
!wdfcommonbuffer 0x57ffa67c988 Size 512
Current: !xhci_transferring 0xfffffa8005b2fe00
PendingTransferList:
[0] dt _TRANSFER_DATA 0xfffffa8005b961b0 !urb
0xfffffa8005b52be8 !wdfrequest 0x57ffa469fd8 TransferState_Pending
```

Now you can pass the address of the transfer ring to the [!xhci_transferring](#) command.

```
dbgcmd

kd> !xhci_transferring 0xfffffa8005b2fe00

[ 0] NORMAL      0x000000011692fe00 CycleBit 1 IOC 0 BEI 0
InterrupterTarget 0 TransferLength    13 TDSIZE 0
[ 1] EVENT_DATA  0x000000011692fe10 CycleBit 1 IOC 1 BEI 0
InterrupterTarget 0 Data  0xfffffa8005986850 TotalBytes 13
[ 2] NORMAL      0x000000011692fe20 CycleBit 1 IOC 0 BEI 0
InterrupterTarget 0 TransferLength    13 TDSIZE 0
[ 3] EVENT_DATA  0x000000011692fe30 CycleBit 1 IOC 1 BEI 0
```

```
InterrupterTarget 0 Data 0 0xfffffa8005b96210 TotalBytes 13
    [ 4] NORMAL      0x000000011692fe40 CycleBit 1 IOC 0 BEI 0
InterrupterTarget 0 TransferLength 13 TDSize 0
    [ 5] EVENT_DATA 0x000000011692fe50 CycleBit 1 IOC 1 BEI 0
InterrupterTarget 0 Data 0 0xfffffa8005b96210 TotalBytes 13
```

See also

[USB 3.0 Extensions](#)

[!xhci_dumpall](#)

[Universal Serial Bus \(USB\) Drivers](#)

USB 2.0 Extensions

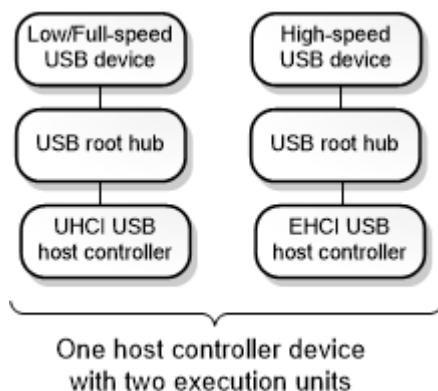
Article • 10/25/2023

This section describes the USB 2.0 debugger extension commands. These commands display information from data structures maintained by drivers in the USB 2.0 driver stack. For more information about these three drivers, see [USB host-side drivers in Windows](#).

The USB 2.0 debugger extension commands are implemented in Usbkd.dll. To load the Usbkd commands, enter `.load usbkd.dll` in the debugger.

USB 2.0 Tree

The USB 2.0 tree contains the device nodes that represent execution units on EHCI host controller devices along with the child nodes that represent hubs and connected devices. This diagram shows an example of a USB 2.0 tree.



The diagram shows one physical host controller device that has two execution units. Each execution unit appears as a device node in the Plug and Play device tree. One execution unit appears as a UHCI USB host controller node, and the other execution unit shows as an EHCI USB host controller node. Each of those nodes has a child node that represents a USB root hub. Each root hub has a single child node that represents a connected USB device.

Notice that the diagram is not a tree in the sense that not all nodes descend from a single parent node. However, when we use the term *USB 2.0 tree*, we are referring to the set of device nodes that represent execution units on EHCI host controller devices along with the nodes for hubs and connected devices.

Getting started with USB 2.0 debugging

To start debugging a USB 2.0 issue, enter the [!usb2tree](#) command. The [!usb2tree](#) command displays a list of commands and addresses that you can use to investigate host controllers, hubs, ports, devices, endpoints, and other elements of the USB 2.0 tree.

In this section

- [!usbkd.usbhelp](#)
- [!usbkd._ehcidd](#)
- [!usbkd._ehciep](#)
- [!usbkd._ehciframe](#)
- [!usbkd._ehciqh](#)
- [!usbkd._ehciregs](#)
- [!usbkd._ehcisitd](#)
- [!usbkd._ehcistq](#)
- [!usbkd._ehcitd](#)
- [!usbkd._ehcifter](#)
- [!usbkd._ehciitd](#)
- [!usbkd.doesdumphaveusbdata](#)
- [!usbkd.isthisdumpasyncissue](#)
- [!usbkd.urbfunc](#)
- [!usbkd.usb2](#)
- [!usbkd.usb2tree](#)
- [!usbkd.usbchain](#)
- [!usbkd.usbdevobj](#)
- [!usbkd.usbdpc](#)
- [!usbkd.ehci_info_from_fdo](#)
- [!usbkd.usbdevh](#)
- [!usbkd.usbep](#)
- [!usbkd.usbfaildata](#)
- [!usbkd.usbhcdext](#)
- [!usbkd.usbdstatus](#)
- [!usbkd.usbhcdhccontext](#)
- [!usbkd.usbhcdlist](#)
- [!usbkd.usbhcdlistlogs](#)
- [!usbkd.usbhcdlog](#)
- [!usbkd.usbhcdlogex](#)
- [!usbkd.usbhcdpnp](#)
- [!usbkd.usbhcdpow](#)
- [!usbkd.hub2_info_from_fdo](#)
- [!usbkd.usbhuberr](#)

- [!usbkd.usbhubext](#)
- [!usbkd.usbhubinfo](#)
- [!usbkd.usbhublog](#)
- [!usbkd.usbhubmddevext](#)
- [!usbkd.usbhubmdpd](#)
- [!usbkd.usbhubpd](#)
- [!usbkd.usbhubs](#)
- [!usbkd.usblist](#)
- [!usbkd.usbpo](#)
- [!usbkd.usbpdos](#)
- [!usbkd.usbpdoxls](#)
- [!usbkd.usbpnp](#)
- [!usbkd.usbportisasyncadv](#)
- [!usbkd.usbportmdportlog](#)
- [!usbkd.usbportmddcontext](#)
- [!usbkd.usbportmddevext](#)
- [!usbkd.usbtriage](#)
- [!usbkd.usbtt](#)
- [!usbkd.usbtx](#)
- [!usbkd.usbusb2ep](#)
- [!usbkd.usbusb2tt](#)
- [!usbkd.usbver](#)

See also

[USB 3.0 Extensions](#)

[RCDRKD Extensions](#)

!usbkd.usbhelp

Article • 10/25/2023

The **!usbkd.usbhelp** command displays help for the USB 2.0 debugger extension commands.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcidd

Article • 10/25/2023

The **!usbkd._ehcidd** command displays information from a **usbehci!_DEVICE_DATA** structure.

```
dbgcmd
```

```
!usbkd._ehcidd StructAddr
```

Parameters

StructAddr

Address of a **usbehci!_DEVICE_DATA** structure. To find addresses of **usbehci!_DEVICE_DATA** structures, use [!usbhcdext](#) or [!usbhcdlist](#).

DLL

Usbkd.dll

Examples

Here is one way to get the address of a **usbehci!_DEVICE_DATA** structure. First enter [!usbkd.usbhcdlist](#).

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdlist
MINIPORT List @ fffff80001e5bbd0
## List of EHCI controllers
!drvobj fffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER
fffffe00001f48bd0 Registration Packet fffffe00001f48c08

01. Xxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
!devobj fffffe0000781a050
!ehci_info fffffe0000781a1a0
Operational Registers fffffd00021fb8420
Device Data fffffe0000781bda0
...
```

In the preceding output, `fffffe0000781bda0` is the address of a `_DEVICE_DATA` structure.

Now pass the structure address to `!ehcidd`

```
dbgcmd

0: kd> !usbkd._ehcidd fffffe0000781bda0

*USBEHCI DEVICE DATA fffffe0000781bda0
** dt usbehci!_DEVICE_DATA fffffe0000781bda0

get_field_ulong fffffe0000781bda0 usbehci!_DEVICE_DATA Flags
*All Endpoints list:
head @ fffffe0000781bdb0 f_link fffffe0000781bdb0 b_link fffffe0000781bdb0
AsyncQueueHead fffffd00021cf5000 !_ehcikh fffffd00021cf5000
    PhysicalAddress: 0xde79a000
    NextQh: fffffd00021cf5000 Hlink de79a002
    PrevQh: fffffd00021cf5000
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcied

Article • 10/25/2023

The **!usbkd._ehcied** command displays information from a **usbehci!_ENDPOINT_DATA** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

```
dbgcmd
```

```
!usbkd._ehcied StructAddr
```

Parameters

StructAddr

Address of a **usbehci!_ENDPOINT_DATA** structure. To find addresses of **usbehci!_ENDPOINT_DATA** structures, use [!usbhcdext](#) and [!usblist](#).

DLL

Usbkd.dll

Examples

This example shows one way to get the address of a **usbehci!_ENDPOINT_DATA** structure. Start with the [!usb2tree](#) command.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe0000206e1a0 !devobj fffffe0000206e050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000024a61a0 !devstack fffffe000024a6050
        Port 1: !port2_info fffffe000026dd000
        Port 2: !port2_info fffffe000026ddb40
        Port 3: !port2_info fffffe000026de680 !devstack fffffe00001ec3060
            !device2_info fffffe00001ec31b0 (USB Mass Storage Device: Xxx
Corporation)
        Port 4: !port2_info fffffe000026df1c0
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command `!ehci_info fffffe0000206e1a0`. Either click the DML command or pass the address of the device extension to `!usbhcdest`.

```
dbgcmd

0: kd> !usbkd.usbhcdest fffffe0000206e1a0
...
DeviceHandleList: !usblist fffffe0000206f3b8, DL
DeviceHandleDeletedList: !usblist fffffe0000206f3c8, DL [Empty]
GlobalEndpointList: !usblist fffffe0000206f388, EP
...
```

The preceding output displays the command `!usblist fffffe0000206f388, EP`. Use this command to display a list of endpoints.

```
dbgcmd

0: kd> !usblist fffffe0000206f388, EP
list: fffffe0000206f388 EP
...
dt usbport!_HCD_ENDPOINT fffffe000026dc970 !usbep fffffe000026dc970
common buffer bytes 0x00003000 (12288) @ va 0000000021e6e000 pa
00000000d83c9000
Device Address: 0x01, ep 0x81 Bulk In Flags: 00000041 dt _USB_ENDPOINT_FLAGS
fffffe000026dc990
... usbehci!_ENDPOINT_DATA fffffe000026dcc38
...
```

In the preceding output, `fffffe000026dcc38` is the address of a `usbehci!_ENDPOINT_DATA` structure. Pass this address to `!ehciep`.

```
dbgcmd

0: kd> !usbkd._ehciep fffffe000026dcc38
*USBEHCI
dt usbehci!_ENDPOINT_DATA fffffe000026dcc38
Flags: 0x00000000
dt usbehci!_HCD_QUEUEHEAD_DESCRIPTOR fffffd00021e6e080
*HwQH fffffd00021e6e080
HwQH
    HwQH.HLink dea2e002
    HwQH.EpChars 02002101
        DeviceAddress: 0x1
        IBit: 0x0
        EndpointNumber: 0x1
        EndpointSpeed: 0x2 HcEPCHAR_HighSpeed
        DataToggleControl: 0x0
        HeadOfReclamationList: 0x0
```

```
MaximumPacketLength: 0x200 - 512
...
current slot 0000000000000000
slot[0] dt usbehci!_ENDPOINT_SLOT fffffe000026dcdb8 - slot_NotBusy
-----
    fffffd00021e6e100
    dt usbehci!_HCD_TRANSFER_DESCRIPTOR fffffd00021e6e100
    tdphys: d83c9100'200 txlen 00000000 tx fffffd00000000041 flags 6d4e695d
_BUSY _SLOT_RESET
    Next_qTD: d83c9200'd83c9180 AltNext_qTD 77423c00'41
    NextTD: fffffd00021e6e200 AltNextTD fffffd00021e6e180 SlotNextTd
fffffd00021e6e200 tok 00000c00 Xbytes x0 (0)
.
    fffffd00021e6e200
    dt usbehci!_HCD_TRANSFER_DESCRIPTOR fffffd00021e6e200
    tdphys: d83c9200'5000 txlen 00000000 tx fffffd00000000041 flags 6d4e695d
_BUSY _SLOT_RESET
    Next_qTD: d83c9280'd83c9180 AltNext_qTD 77423c00'41
    NextTD: fffffd00021e6e280 AltNextTD fffffd00021e6e180 SlotNextTd
fffffd00021e6e280 tok 00000c00 Xbytes x0 (0)
...

```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehciframe

Article • 10/25/2023

The **!usbkd._ehciframe** command displays an EHCI miniport FrameListBaseAddress periodic list entry chain indexed by a frame number.

dbgcmd

```
!usbkd._ehciframe StructAddr, FrameNumber
```

Parameters

StructAddr

Address of a **usbehci!_DEVICE_DATA** structure.

FrameNumber

Frame number in the range 0 through 1023.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehciqh

Article • 10/25/2023

The **!usbkd._ehciqh** command displays information from a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

dbgcmd

```
!usbkd._ehciqh StructAddr
```

Parameters

StructAddr

Address of a **usbehci!_HCD_QUEUEHEAD_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehciregs

Article • 10/25/2023

The **!usbkd._ehciregs** command displays the operational and root hub port status registers of a USB EHCI host controller.

dbgcmd

```
!usbkd._ehciregs StructAddr[, NumPorts]
```

Parameters

StructAddr

Address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure. To find the address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure, use [!usbkd.usbhcdlist](#).

NumPorts

The number of root hub port status registers to display.

DLL

Usbkd.dll

Examples

Here is one way to get the address of a **usbehci!_HC_OPERATIONAL_REGISTER** structure. First enter [!usbkd.usbhcdlist](#).

dbgcmd

```
0: kd> !usbkd.usbhcdlist
MINIPORT List @ fffff80001e5bbd0

## List of EHCI controllers

!drvobj fffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER ...
...
02. Xxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
    !devobj fffffe00001ca1050
    !ehci_info fffffe00001ca11a0
    Operational Registers fffffd000228bf020
```

In the preceding output, `fffffd000228bf020` is the address of a `_HC_OPERATIONAL_REGISTER` structure.

Now pass the structure address to `!ehciregs`. In this example, the second argument limits the display to two root hub port status registers.

```
dbgcmd
```

```
0: kd> !usbkd._ehciregs fffffd000228bf020, 2
*(ehci)HC_OPERATIONAL_REGISTER fffffd000228bf020
    USBCMD 00010001
    .HostControllerRun: 1
    .HostControllerReset: 0
    .FrameListSize: 0
    .PeriodicScheduleEnable: 0
    .AsyncScheduleEnable: 0
    .IntOnAsyncAdvanceDoorbell: 0
    .HostControllerLightReset: 0
    .InterruptThreshold: 1
    .ParkModeEnable: 0
    .ParkModeCount: 0

    USBSTS 00002008
    .UsbInterrupt: 0
    .UsbError: 0
    .PortChangeDetect: 0
    .FrameListRollover: 1
    .HostSystemError: 0
    .IntOnAsyncAdvance: 0
    ----
    .HcHalted: 0
    .Reclamation: 1
    .PeriodicScheduleStatus: 0
    .AsyncScheduleStatus: 0

    USBINTR 0000003f
    .UsbInterrupt: 1
    .UsbError: 1
    .PortChangeDetect: 1
    .FrameListRollover: 1
    .HostSystemError: 1
    .IntOnAsyncAdvance: 1
    PeriodicListBase dec8e000
    AsyncListAddr dec91000
    PortSC[0] 00001000
        PortConnect x0
        PortConnectChange x0
        PortEnable x0
        PortEnableChange x0
        OvercurrentActive x0
        OvercurrentChange x0
        ForcePortResume x0
        PortSuspend x0
```

```
PortReset x0
HighSpeedDevice x0
LineStatus x0
PortPower x1
PortOwnedByCC x0
PortIndicator x0
PortTestControl x0
WakeOnConnect x0
WakeOnDisconnect x0
WakeOnOvercurrent x0
PortSC[1] 00001000
    PortConnect x0
    PortConnectChange x0
    PortEnable x0
    PortEnableChange x0
    OvercurrentActive x0
    OvercurrentChange x0
    ForcePortResume x0
    PortSuspend x0
    PortReset x0
    HighSpeedDevice x0
    LineStatus x0
    PortPower x1
    PortOwnedByCC x0
    PortIndicator x0
    PortTestControl x0
    WakeOnConnect x0
    WakeOnDisconnect x0
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcisitd

Article • 10/25/2023

The `!usbkd._ehcisitd` command displays information from a `usbehci!_HCD_SI_TRANSFER_DESCRIPTOR`

dbgcmd

```
!usbkd._ehcisitd StructAddr
```

Parameters

StructAddr

Address of a `usbehci!_HCD_SI_TRANSFER_DESCRIPTOR` structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcistq

Article • 10/25/2023

The `!usbkd._ehcistq` command displays a `usbehci!_HCD_QUEUEHEAD_DESCRIPTOR` structure.

dbgcmd

```
!usbkd._ehcierp StructAddr
```

Parameters

StructAddr

Address of a `usbehci!_HCD_QUEUEHEAD_DESCRIPTOR` structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcitd

Article • 10/25/2023

The **!usbkd._ehcitd** command displays information from a **usbehci!_TRANSFER_CONTEXT** structure. Use this command to display information about asynchronous endpoints (that is, control and bulk endpoints).

```
dbgcmd
```

```
!usbkd._ehcitd StructAddr
```

Parameters

StructAddr

Address of a **usbehci!_TRANSFER_CONTEXT** structure.

DLL

Usbkd.dll

Examples

This example shows one way to get the address of a **usbehci!_TRANSFER_CONTEXT** structure. Use [!**ehciep**](#) to display information about an endpoint.

```
dbgcmd
```

```
0: kd> !_ehciep fffffe000001ab618
*USBEHCI
dt usbehci!_ENDPOINT_DATA fffffe000001ab618
Flags: 0x00000000
dt usbehci!_HCD_QUEUEHEAD_DESCRIPTOR fffffd00021e65080
*HwQH fffffd00021e65080
HwQH
    HwQH.HLink dea2e002
    HwQH.EpChars 02002201
        DeviceAddress: 0x1
        IBit: 0x0
        EndpointNumber: 0x2
    ...
slot[0] dt usbehci!_ENDPOINT_SLOT fffffe000001ab798 - slot_NotBusy
-----
fffffd00021e65100
```

```
dt usbehci!_HCD_TRANSFER_DESCRIPTOR fffffd00021e65100
....
```

In the preceding output, `fffffd00021e65100` is the address of a `usbehci!TRANSFER_CONTEXT` structure. Pass this address to `!ehcitr`.

dbgcmd

```
0: kd> !ehcitr fffffd00021e65100
*USBEHCI TD 21e65100
Sig 20td
    qTD
    Next_qTD: d83cc200
    AltNext_qTD: d83cc180
    Token: 0x00000c00
        PingState: 0x0
        SplitXstate: 0x0
        MissedMicroFrame: 0x0
        XactErr: 0x0
        BabbleDetected: 0x0
        DataBufferError: 0x0
        Halted: 0x0
        Active: 0x0
        Pid: 0x0 - HcTOK_Out
        ErrorCounter: 0x3
        C_Page: 0x0
        InterruptOnComplete: 0x0
        BytesToTransfer: 0x0
        DataToggle: 0x0
        BufferPage[0]: 0x 0bad0-000 0bad0000 BufferPage64[0]: 00000000
        BufferPage[1]: 0x 0bad0-000 0bad0000 BufferPage64[1]: 00000000
        BufferPage[2]: 0x 0bad0-000 0bad0000 BufferPage64[2]: 00000000
        BufferPage[3]: 0x 0bad0-000 0bad0000 BufferPage64[3]: 00000000
        BufferPage[4]: 0x 0bad0-000 0bad0000 BufferPage64[4]: 00000000
    Packet:00 52 e6 21 00 d0 ff ff
    PhysicalAddress: d83cc100
    EndpointData: 001ab618
    TransferLength : 0000001f
    TransferContext: 00000000
    Flags: 00000041
        TD_FLAG_BUSY
    NextHcdTD: 21e65200
    AltNextHcdTD: 21e65180
    SlotNextHcdTD: 21e65200
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehcitfer

Article • 10/25/2023

The **!usbkd._ehcitfer** command displays information from a **usbehci!_HCD_TRANSFER_DESCRIPTOR** structure.

dbgcmd

```
!usbkd._ehcitfer StructAddr
```

Parameters

StructAddr

Address of a **usbehci!_HCD_TRANSFER_DESCRIPTOR** structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd._ehciitd

Article • 10/25/2023

The `!usbkd._ehciitd` command displays information from a `usbehci!_HCD_HSISO_TRANSFER_DESCRIPTOR` structure.

dbgcmd

```
!usbkd._ehciitd StructAddr
```

Parameters

StructAddr

Address of a `usbehci!_HCD_HSISO_TRANSFER_DESCRIPTOR` structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.doesdump haveusbdata

Article • 10/25/2023

The `!usbkd.doesdump haveusbdata` command checks to see which types of USB data are in a crash dump file that was generated as a result of [Bug Check 0xFE](#).

```
dbgcmd
```

```
!usbkd.doesdump haveusbdata
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

Examples

Here is an example of the output of `!doesdump haveusbdata`

```
dbgcmd
```

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.doesdump haveusbdata
```

```
Retrieving crashdump information Please Wait...
```

```
Checking for GuidUsbHubPortArrayData information...
There is no data for this GUID in the mini dump.
No data to print
```

```
Checking for GuidUsbHubExt information...
There is no data for this GUID in the mini dump.
No data to print
```

```
Checking for GuidUsbPortLog information...
GuidUsbPortLog Exists with PORT Log Size = 8000
```

```
Checking for GuidUsbPortContextData information...
GuidUsbPortContextData Exists with Data Length = 4c8
```

```
Checking for GuidUsbPortExt information...
GuidUsbPortExt Exists (DEVICE_EXTENSION + DeviceDataSize ) = 2250
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.isthisdumpasyncissue

Article • 10/25/2023

The `!usbkd.isthisdumpasyncissue` command checks a crash dump file, generated by [Bug Check 0xFE](#), to see whether the likely cause of the crash was an Interrupt on Async Advance issue associated with a USB EHCI host controller.

```
dbgcmd
```

```
!usbkd.isthisdumpasyncissue
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

Examples

Here is an example of the output of `!usbkd.isthisdumpasyncissue`.

```
dbgcmd
```

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.isthisdumpasyncissue
This is *NOT* Async on Advance Issue because the EndPointData is NULL
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.urbfunc

Article • 10/25/2023

The **!usbkd.urbfunc** command displays the name of a URB function code.

```
dbgcmd
```

```
!usbkd.urbfunc FunctionCode
```

Parameters

FunctionCode

The hexadecimal value of a URB function code. These codes are defined in `usb.h`.

DLL

`Usbkd.dll`

Examples

Here is an example of the output of **!urbfunc**.

```
dbgcmd
```

```
0: kd> !usbkd.urbfunc 0xA
```

```
URB_FUNCTION_ISOCH_TRANSFER (0xA)
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usb2

Article • 10/25/2023

The **!usbkd.usb2** command displays a list of USB endpoints that have USB 2.0 scheduling information.

```
dbgcmd
```

```
!usbkd.usb2 DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a USB host controller.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command `!ehci_info fffffe00001ca11a0`. Pass the address of the device extension to the **!usb2** command.

```
dbgcmd
```

```
0: kd> !usbkd.usb2 fffffe00001ca11a0

Sig: HFDO
Hcd FDO Extension:
-----
-----
dt usbport!_HCD_ENDPOINT fffffe0000212d970 !usbep fffffe0000212d970
    Tt 0000000000000000 Device Address: 0x00, ep 0x81 Interrupt In
    dt _USB2LIB_ENDPOINT_CONTEXT fffffe000023b60f0      dt _USB2_EP
fffffe000023b6100
    Period,offset,Ordinal(32,0,0)    smask,cmask(00,00 ..... , ....)
maxpkt 1
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usb2tree

Article • 10/25/2023

The **!usbkd.usb2tree** command displays [USB 2.0 tree](#).

```
dbgcmd
```

```
!usbkd.usb2tree
```

Examples

This screen shot shows and example of the output of the **!usb2tree** command.

```

!usbkd.usb2tree - Kernel 'net:port=50002,key=*****' - WinDbg:6.13...
Command: !usbkd.usb2tree
UHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48010 ^
1)!uhci_info fffffe00001ca41a0 !devobj fffffe00001ca4050 PCI: VendorId 80
RootHub !hub2_info fffffe0000231c1a0 !devstack fffffe0000231c050
Port 1: !port2_info fffffe0000212e000
Port 2: !port2_info fffffe0000212eb40 !devstack fffffe000061342a0
!device2_info fffffe000061343f0 (Generic Bluetooth Radio: Ca

2)!uhci_info fffffe00001ca71a0 !devobj fffffe00001ca7050 PCI: VendorId 80
RootHub !hub2_info fffffe000023141a0 !devstack fffffe00002314050
Port 1: !port2_info fffffe0000212c000
Port 2: !port2_info fffffe0000212cb40

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0
1)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 80
RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
Port 1: !port2_info fffffe000021bf000
Port 2: !port2_info fffffe000021bfb40
Port 3: !port2_info fffffe000021c0680 !devstack fffffe000037691a0
!device2_info fffffe000037692f0 (USB Mass Storage Device: Sa
Port 4: !port2_info fffffe000021c11c0

Enumerated HUB List
-----
Root Hubs:
1) FDO fffffe0000231c050 PDO fffffe0000213a050 Depth 0
!hub2_info fffffe0000231c1a0
Parent HC: !uhci_info fffffe00001ca41a0
FDO Power State: FdoS0_D0

2) FDO fffffe00002314050 PDO fffffe00002140050 Depth 0
!hub2_info fffffe000023141a0
Parent HC: !uhci_info fffffe00001ca71a0
FDO Power State: FdoS0_D0

3) FDO fffffe00002320050 PDO fffffe0000213c050 Depth 0
!hub2_info fffffe000023201a0
Parent HC: !ehci_info fffffe00001ca11a0
FDO Power State: FdoS0_D0

Downstream Hubs:
-----
Enumerated Device List
-----
1) :USB\VID_0A12&PID_0001&REV_0309USB\VID_0A12&PID_0001
Vendor: Cambridge Silicon Radio Ltd.
Generic Bluetooth Radio
!device2_info fffffe000061343f0 !devstack fffffe000061342a0
Device Descriptor fffffe00006134960
Parent Hub: !hub2_info fffffe0000231c1a0
PDO Hw PnP State: Pdo_PnpRefHwPresent
PDO Power State: Pdo_D0

2) :USB\VID_0781&PID_5530&REV_0100USB\VID_0781&PID_5530
Vendor: SanDisk Corporation
USB Mass Storage Device
!device2_info fffffe000037692f0 !devstack fffffe000037691a0
Device Descriptor fffffe00003769860
Parent Hub: !hub2_info fffffe000023201a0
PDO Hw PnP State: Pdo_PnpRefHwPresent
PDO Power State: Pdo_D0

```

The output shows one EHCI execution unit and two UHCI execution units. The execution units shown in this example happen to be on a single USB host controller device. The output also shows the root hubs and connected devices.

The output uses [Using Debugger Markup Language \(DML\)](#) to provide links. The links execute commands that give detailed information related to objects in the tree. For example, you could click one of the `!devobj` links to get information about the functional device object associated with the EHCI execution unit. As an alternative to clicking the link, you could enter the command manually: `!devobj fffffe00001ca7050`

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Usb3kd.dll

Remarks

The `!usb2tree` command is the parent command for many of the [USB 2.0 debugger extensions commands](#). The information displayed by these commands is based on data structures maintained by these drivers:

- usbehci.sys (miniport driver for USB 2 host controller)
- usbuhci.sys (miniport driver for USB 2 host controller)
- usbport.sys (port driver for USB 2 host controller)
- usbhub.sys (USB 2 hub driver)

For more information about these drivers, see [USB host-side drivers in Windows](#).

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbchain

Article • 10/25/2023

The **!usbkd.usbchain** command displays a USB device chain starting at a specified PDO, and going back to the root hub.

```
dbgcmd
```

```
!usbkd.usbchain PDO
```

Parameters

PDO

Address of the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the PDO of a USB device. First enter

!usbkd.usb2tree.

```
dbgcmd
```

```
kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
        Port 1: !port2_info fffffe000021bf000
        Port 2: !port2_info fffffe000021fbfb40
        Port 3: !port2_info fffffe000021c0680 !devstack fffffe00007c882a0
...
```

In the preceding output, the address of the PDO is the argument of the suggested command **!devstack fffffe00007c882a0**. Pass the address of the PDO to **!usbkd.usbchain**.

```
dbgcmd
```

```
0: kd> !usbkd.usbchain fffffe00007c882a0

usbchain
*****
*
HUB PDO fffffe00007c882a0 on port 3 !usbhubext fffffe00007c883f0 ArmedForWake
= 0
VID XXXX PID XXXX REV 0100 XXXX Corporation
    HUB #3 FDO fffffe00002320050 , !usbhubext fffffe000023201a0 HWC_ARM=0
    ROOT HUB PDO(ext) @fffffe0000213c1a0
        ROOT HUB FDO @fffffe00001ca1050, !usbhcdest fffffe00001ca11a0 PCI
Vendor:Device:...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbdevobj

Article • 10/25/2023

The `!usbkd.usbdevobj` command displays information from a USB device object.

dbgcmd

```
!usbkd.usbdevobj DeviceObject
```

Parameters

DeviceObject

Address of a `_DEVICE_OBJECT` structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbdpc

Article • 10/25/2023

The **!usbkd.usbdpc** command displays information stored in an **_XDPC_CONTEXT** structure.

```
dbgcmd
```

```
!usbkd.usbdpc StructAddr
```

Parameters

StructAddr

Address of a **usbport!_XDPC_CONTEXT** structure. To get the XDPC list for a USB host controller, use the [!usbkd.usbhcdext](#) command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_XDPC_CONTEXT** structure. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
UHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001e77010
...
4)!uhci_info fffffe00001c7d1a0 !devobj fffffe00001c7d050 PCI: VendorId...
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command [!uhci_info fffffe00001c7d1a0](#).

Either click the DML command or pass the address of the device extension to [!usbhcdext](#) to get the XDPC list.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdest fffffe00001c7d1a0
...
## XDPC List

01) dt USBPORT!_XDPC_CONTEXT fffffe00001c7df18
02) dt USBPORT!_XDPC_CONTEXT fffffe00001c7db88
03) dt USBPORT!_XDPC_CONTEXT fffffe00001c7dd50
04) dt USBPORT!_XDPC_CONTEXT fffffe00001c7e0e0
...
```

In the preceding output, `fffffe00001c7df18` is the address of an `_XDPC_CONTEXT` structure. Pass this address to `!usbpdpc`.

```
dbgcmd

0: kd> !usbkd.usbpdpc fffffe00001c7df18
dt USBPORT!_XDPC_CONTEXT fffffe00001c7df18
## XDPC HISTORY (latest at boottom)

##      EVENT          STATE          NEXT
[01] Ev_Xdpc_End    XDPC_Running   XDPC_Enabled
[02] Ev_Xdpc_Signal XDPC_Enabled   XDPC_DpcQueued
[03] Ev_Xdpc_Signal XDPC_DpcQueued XDPC_DpcQueued
[04] Ev_Xdpc_Worker XDPC_DpcQueued XDPC_Running
[05] Ev_Xdpc_Signal XDPC_Running   XDPC_Signaled
[06] Ev_Xdpc_End    XDPC_Signaled  XDPC_DpcQueued
[07] Ev_Xdpc_Worker XDPC_DpcQueued XDPC_Running
[08] Ev_Xdpc_End    XDPC_Running   XDPC_Enabled
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.ehci_info_from_fdo

Article • 10/25/2023

The [!usbkd.ehci_info_from_fdo](#) command displays information about a USB host controller.

```
dbgcmd
```

```
!usbkd.ehci_info_from_fdo fdo
```

Parameters

fdo

Address of the functional device object (FDO) of a UHCI or EHCI USB host controller. You can get the address of the FDO from the output of the [!usb2tree](#) command.

DLL

Usbkd.dll

Examples

First use the [!usb2tree](#) command to get the address of the FDO.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0

1)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, you can see that the address of the FDO of the USB host controller is `fffffe00001ca1050`. Pass the address of the FDO to [!ehci_info_from_fdo](#).

```
dbgcmd
```

```
0: kd> !usbkd.ehci_info_from_fdo fffffe00001ca1050

HC Flavor 1000 FDO fffffe00001ca1050
```

```
Root Hub: FDO fffffe00002320050 !hub2_info fffffe000023201a0
Operational Registers fffffd000228bf020
Device Data fffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION fffffe00001ca15a0
DM Timer Flags fffffe00001ca16d4
FDO Flags fffffe00001ca16d0
HCD Log fffffe00001ca11a0

DeviceHandleList: !usblist fffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist fffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist fffffe00001ca2388, EP
EpNeoStateChangeList: !usblist fffffe00001ca2370, SC [Empty]
GlobalTtListHead: !usblist fffffe00001ca23a8, TT [Empty]
BusContextHead: !usblist fffffe00001ca16b0, BC

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT fffffe00001ca1450 Tag: AddD Obj:
fffffe00001ca11a0
...
## XDPC List

01) dt USBPORT!_XDPC_CONTEXT fffffe00001ca1f18
...
## PnP FUNC HISTORY (latest at bottom)

[01] IRP_MN_QUERY_CAPABILITIES
...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbdevh

Article • 10/25/2023

The **!usbkd.usbdevh** command displays information about a USB device handle.

```
dbgcmd
```

```
!usbkd.usbdevh StructAddr
```

Parameters

StructAddr

Address of a **usbport!_USBD_DEVICE_HANDLE** structure. To get the device handle list for a USB host controller, use the [!usbkd.usbhcdext](#) command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_USBD_DEVICE_HANDLE** structure.

First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command **!ehci_info fffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to [!usbhcdext](#) to get the device handle list.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdext fffffe00001ca11a0

HC Flavor 1000  FDO fffffe00001ca1050
Root Hub: FDO fffffe00002320050 !hub2_info fffffe000023201a0
Operational Registers fffffd000228bf020
Device Data fffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION fffffe00001ca15a0
DM Timer Flags fffffe00001ca16d4
FDO Flags fffffe00001ca16d0
HCD Log fffffe00001ca11a0

DeviceHandleList: !usblist fffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist fffffe00001ca23c8, DL [Empty]
...
```

Now use the **!usbkd.usblist** command to get the addresses of **usbport!_USBD_DEVICE_HANDLE** structures.

```
dbgcmd

0: kd> !usblist fffffe00001ca23b8, DL
list: fffffe00001ca23b8 DL
-----
!usbdevh fffffe000020f9590
SSP [IdleReady] (0)
...
```

In the preceding output, **fffffe000020f9590** is the address of a **_USBD_DEVICE_HANDLE** structure. Pass this address to **!usbdevh**.

```
dbgcmd

0: kd> !usbkd.usbdevh fffffe000020f9590

dt USBPORT!_USBD_DEVICE_HANDLE fffffe000020f9590
SSP [IdleReady] (0)
PCI\VEN_8086&DEV_293C Intel Corporation
Root Hub
DriverName :

## DEVICE HANDLE HISTORY (latest at bottom)

##      EVENT          STATE          NEXT
[01] Ev_CreateRoothub_Success    Devh_Created        Devh_Valid
## Reference List: Head(fffffe000020f9668)

[00] dt USBPORT!_DEVH_REF_OBJ fffffe000021944a0 Object: fffffe000020f9590
Tag: dvh+ PendingFlag(0)
```

```
[01] dt USBPORT!_DEVH_REF_OBJ fffffe000020bbcb0 Object: fffffe000020ba7e0
Tag: bsCT PendingFlag(0)
[02] dt USBPORT!_DEVH_REF_OBJ fffffe000032b91a0 Object: fffffe0000269e670
Tag: UrbT PendingFlag(1)

## TtList: Head(fffe000020f9658)

## PipeHandleList: Head(fffe000020f9640)

[00] dt USBPORT!_USBD_PIPE_HANDLE_I fffffe000020f95e0 !usbep fffffe000020f6970
    Device Address: 0x00, Endpoint Address 0x00 Endpoint Type: Control
[01] dt USBPORT!_USBD_PIPE_HANDLE_I fffffe000023bd278 !usbep fffffe0000212d970
    Device Address: 0x00, Endpoint Address 0x81 Endpoint Type: Interrupt
In

Config Information: dt USBPORT!_USBD_CONFIG_HANDLE fffffe000023cd0b0

## Interface List:

[00] dt USBPORT!_USBD_INTERFACE_HANDLE_I fffffe000023bd250
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbep

Article • 10/25/2023

The **!usbkd.usbep** command displays information about a USB endpoint.

```
dbgcmd
```

```
!usbkd.usbep StructAddr
```

Parameters

StructAddr

Address of a **usbport!_HCD_ENDPOINT** structure. To get the endpoint list for a USB host controller, use the [!usbkd.usbhcext](#) command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_HCD_ENDPOINT** structure. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2:!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command **!ehci_info fffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to [!usbhcext](#) to get the global endpoint list.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdext fffffe00001ca11a0
...
DeviceHandleList: !usblist fffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist fffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist fffffe00001ca2388, EP
...
```

Now use the **!usbkd.usblist** command to get the addresses of **_HCD_ENDPOINT** structures.

```
dbgcmd

0: kd> !usblist fffffe00001ca2388, EP

list: fffffe00001ca2388 EP
-----
dt usbport!_HCD_ENDPOINT fffffe000020f6970 !usbep fffffe000020f6970
Device Address: 0x00, ep 0x00 Control Flags: 00000002 dt
_USB_ENDPOINT_FLAGS fffffe000020f6990
dt usbport!_ENDPOINT_PARAMETERS fffffe000020f6b18 RootHub Endpoint
...
```

In the preceding output, `fffffe000020f6970` is the address of an **_HCD_ENDPOINT** structure. Pass this address to **!usbkd.usbep**.

```
dbgcmd

0: kd> !usbep fffffe000020f6970
Device Address: 0x00, Endpoint Address 0x00 Endpoint Type: Control
dt USBPORT!_HCD_ENDPOINT fffffe000020f6970
dt USBPORT!_ENDPOINT_PARAMETERS fffffe000020f6b18
RootHub Endpoint

## Transfer(s) List: (HwPendingListHead)

[EMPTY]

## Endpoint Reference List: (EpRefListHead)

[00] dt USBPORT!_USBOBJ_REF fffffe000021a64a0 Object fffffe000020f6970
Tag:EPop Endpoint:fffffe000020f6970
[01] dt USBPORT!_USBOBJ_REF fffffe000021264a0 Object fffffe000020f95e0
Tag:EPpi Endpoint:fffffe000020f6970

## GEP HISTORY (latest at bottom)

##      EVENT          STATE          NEXT
HwEpState

[01] Ev_gEp_Open      GEp_Init      GEp_Paused
```

ENDPOINT_PAUSE

[02] Ev_gEp_ReqActive

ENDPOINT_ACTIVE

GEp_Paused

GEp_Active

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbfaildata

Article • 10/25/2023

The **!usbkd.usbfaildata** command displays the failure data (if any) stored for a USB device.

```
dbgcmd
```

```
!usbkd.usbfaildata PDO
```

Parameters

PDO

Address of the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the PDO of a USB device. Enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
        Port 1: !port2_info fffffe000021bf000
        Port 2: !port2_info fffffe000021fb40
        Port 3: !port2_info fffffe000021c0680 !devstack fffffe00007c882a0
    ...
...
```

In the preceding output, the address of the PDO appears as the argument of the suggested command **!devstack fffffe00007c882a0**.

Now pass the address of the PDO to **!usbkd.usbfaildata**.

See also

USB 2.0 Debugger Extensions

Universal Serial Bus (USB) Drivers

!usbkd.usbhcdext

Article • 10/25/2023

The **!usbkd.usbhcdext** command displays information from the device extension of a USB host controller or a USB root hub.

```
dbgcmd
```

```
!usbkd.usbhcdext DeviceExtension
```

Parameters

DeviceExtension

Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of an EHCI host controller. First enter **!usbkd.usb2tree**.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0

1)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the **DML** command **!ehci_info fffffe00001ca11a0**.

Now pass the address of the device extension to the [!usbhcdext](#) command.

```
dbgcmd

0: kd> !usbkd.usbhcdext fffffe00001ca11a0

HC Flavor 1000  FDO fffffe00001ca1050
Root Hub: FDO fffffe00002320050 !hub2_info fffffe000023201a0
Operational Registers fffffd000228bf020
Device Data fffffe00001ca2da0
dt USBPORT!_FDO_EXTENSION fffffe00001ca15a0
DM Timer Flags fffffe00001ca16d4
FDO Flags fffffe00001ca16d0
HCD Log fffffe00001ca11a0

DeviceHandleList: !usblist fffffe00001ca23b8, DL
DeviceHandleDeletedList: !usblist fffffe00001ca23c8, DL [Empty]
GlobalEndpointList: !usblist fffffe00001ca2388, EP
EpNeoStateChangeList: !usblist fffffe00001ca2370, SC [Empty]
GlobalTtListHead: !usblist fffffe00001ca23a8, TT [Empty]
BusContextHead: !usblist fffffe00001ca16b0, BC

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT fffffe00001ca1450 Tag: AddD Obj:
fffffe00001ca11a0
...
## XDPC List

01) dt USBPORT!_XDPC_CONTEXT fffffe00001ca1f18
...
```

Here is one way to find the address of the device extension for the PDO of a root hub.

First enter [!usbkd.usb2tree](#).

```
dbgcmd

0: kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0

1)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
...
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command [!devstack fffffe00002320050](#). Use the [!devstack](#) command to find the address of the PDO and the PDO device extension.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe00002320050
  !DevObj          !DrvObj          !DevExt          ObjectName
> fffffe00002320050  \Driver\usbhub   fffffe000023201a0  0000002d
  fffffe0000213c050  \Driver\usbehci  fffffe0000213c1a0  USBPDO-3
  ...
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `fffffe0000213c1a0`.

Now pass the address of the device extension to the [!usbhcdext](#) command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdext fffffe0000213c1a0

Root Hub PDO Extension
Parent HC: FDO fffffe00001ca1050 !ehci_info fffffe00001ca11a0
HUB FDO fffffe00002320050 !hub2_info fffffe000023201a0
dt USBPORT!_PDO_EXTENSION fffffe0000213c5a0

## Pending Requests

[001] dt USBPORT!_USB_IOREQUEST_CONTEXT fffffe0000213c450 Tag: RHcr Obj:
fffffe0000213c1a0
[002] dt USBPORT!_USB_IOREQUEST_CONTEXT fffffe00003ce5800 Tag: iIRP Obj:
fffffe00002182210

## POWER FUNC HISTORY (latest at bottom)

[00] IRP_MN_WAIT_WAKE (PowerSystemHibernate)
...
## PnP STATE LOG (latest at bottom)

##      EVENT          STATE          NEXT
[01] EvPDO_IRP_MN_START_DEVICE    PnpNotStarted    PnpStarted
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbdstatus

Article • 10/25/2023

The **!usbkd.usbdstatus** command displays the name of a USBD status code.

```
dbgcmd
```

```
!usbkd.usbdstatus StatusCode
```

Parameters

StatusCode

The hexadecimal value of a USBD status code. These codes are defined in usb.h.

DLL

Usbkd.dll

Examples

Here is an example of the output of **!usbstatus**.

```
dbgcmd
```

```
1: kd> !usbkd.usbdstatus 0xC0000008
USBD_STATUS_DATA_OVERRUN (0xC0000008)
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdhccontext

Article • 10/25/2023

The **!usbkd.usbhcdhccontext** command displays the **USB2LIB_HC_CONTEXT** for a USB host controller.

dbgcmd

```
!usbkd.usbhcdhccontext DeviceExtension
```

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a USB host controller.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdlist

Article • 10/25/2023

The [!usbkd.usbhcdlist](#) command displays information about all USB host controllers that are represented by the USB port driver (Usbport.sys). For information about the USB port driver and the associated miniport drivers, see [USB host-side drivers in Windows](#).

```
dbgcmd
```

```
!usbkd.usbhcdlist
```

DLL

Usbkd.dll

Examples

Here is an example of a portion of the output of [!usbhcldlist](#).

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdlist
MINIPORT List @ fffff80001e5bbd0

## List of UHCI controllers

!drvobj fffffe00002000060 dt USBPORT!_USBPORT_MINIPORT_DRIVER
fffffe00001f48010 Registration Packet fffffe00001f48048

01
...
## List of EHCI controllers

!drvobj fffffe00001fd33a0 dt USBPORT!_USBPORT_MINIPORT_DRIVER
fffffe00001f48bd0 Registration Packet fffffe00001f48c08

01. Xxxxxx Corporation PCI: VendorID Xxxx DeviceID Xxxx RevisionId 0002
    !devobj fffffe00001ca1050
    !ehci_info fffffe00001ca11a0
    Operational Registers fffffd000228bf020
    Device Data fffffe00001ca2da0
    !usbhcdlog fffffe00001ca11a0
    nt!_KINTERRUPT fffffe000020abe78
```

```
Device Capabilities fffffe00001ca135c
Pending IRP's: 0, Transfers: 0 (Periodic(0), Async(0))
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdlistlogs

Article • 10/25/2023

The **!usbkd.usbhcdlistlogs** command displays a list of all functional device objects (FDOs) associated with the USB port driver (Usbport.sys). The command also displays the complete debug logs for all EHCI Host Controllers.

```
dbgcmd
```

```
!usbkd.usbhcdlistlogs
```

DLL

Usbkd.dll

Examples

This example shows a portion of the output of the **!usbhcldlistlogs** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdlistlogs

MINIPORT List @ fffff80001e5bbd0
*flink, blink fffffe00001f48018,fffffe00001f48bd8

[0] entry @: fffffe00001f48010 dt usbport!_USBPORT_MINIPORT_DRIVER
fffffe00001f48010
UHCI MINIPORT @ fffffe00001f48010 !drvobj fffffe00002000060 regpkt
fffffe00001f48048
02. !devobj fffffe00001ca4050 !usbhcdext fffffe00001ca41a0 HFDO 8086 2938
RHPDO fffffe0000213a050
03. !devobj fffffe00001ca7050 !usbhcdext fffffe00001ca71a0 HFDO 8086 2937
RHPDO fffffe00002140050

[1] entry @: fffffe00001f48bd0 dt usbport!_USBPORT_MINIPORT_DRIVER
fffffe00001f48bd0
EHCI MINIPORT @ fffffe00001f48bd0 !drvobj fffffe00001fd33a0 regpkt
fffffe00001f48c08
01. !devobj fffffe00001ca1050 !usbhcdext fffffe00001ca11a0 HFDO 8086 293c
RHPDO fffffe0000213c050

LOG@: fffffe00001ca11b8
>LOG mask = 3ff idx = fff69e8d (28d)
*LOG: fffffe000020191a0 LOGSTART: fffffe00002014000 *LOGEND: fffffe0000201bfe0
# -1
```

```
[ 000] fffffe000020191a0 Tmt0 0000000000000000 0000000000000000
0000000000000000
[ 001] fffffe000020191c0 _Pop 0000000000000003 0000000000003000
0000000000000000
...
[ 005] fffffe00002019240 chgZ 0000000000000000 000000000b7228f
0000000000000000
[ 006] fffffe00002019260 nes- 0000000000000000 0000000000000000
0000000000000000
[ 007] fffffe00002019280 nes+ fffffe00001ca2370 fffffe00001ca2370
0000000000ced39
...
[1014] fffffe00002019060 tmo4 0000000000000000 0000000000000040
fffffe00001ca1c20
...
[1022] fffffe00002019160 xdw2 fffffe00001ca1b88 fffffe00001ca1050
0000000000000002
[1023] fffffe00002019180 xdB0 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
```

The command output shows two FDOs that represent UHCI host controllers and one FDO that represents an EHCI host controller. Then the output shows the debug log for the one EHCI host controller.

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdlog

Article • 10/25/2023

The [!usbkd.usbhcdlog](#) command displays a portion of the debug log for a USB host controller.

```
dbgcmd
```

```
!usbkd.usbhcdlog DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a UHCI or EHCI USB host controller.

NumberOfEntries

The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0 kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command `!ehci_info fffffe00001ca11a0`.

Now pass the address of the device extension to the **!usbhcdlog** command. In this example, the second argument limits the display to four log entries.

```
dbgcmd

0: kd> !usbkd.usbhcdlog fffffe00001ca11a0, 4

LOG@: fffffe00001ca11b8
>LOG mask = 3ff idx = fff68e95 (295)
*LOG: fffffe000020192a0 LOGSTART: fffffe00002014000 *LOGEND: fffffe0000201bfe0
# 4
[ 000] fffffe000020192a0 xSt0 fffffe00001ca1b88 0000000000000006
0000000000000001
[ 001] fffffe000020192c0 xnd8 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
[ 002] fffffe000020192e0 xnd0 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
[ 003] fffffe00002019300 gNX0 0000000000000000 0000000000000000
fffffe00001ca1b88
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcldlogex

Article • 10/25/2023

The `!usbkd.usbhcldlogex` command displays an annotated debug log for a USB host controller.

```
dbgcmd
```

```
!usbkd.usbhcldlogex DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a UHCI or EHCI USB host controller.

NumberOfEntries

The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB host controller. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0 kd> !usbkd.usb2tree

EHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe00001f48bd0
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command `!ehci_info fffffe00001ca11a0`.

Now pass the address of the device extension to the !usbhcdlogex command. In this example, the second argument limits the display to 20 log entries.

```
dbgcmd

0: kd> !usbkd.usbhcdlogex fffffe00001ca11a0, 20
LOG@: fffffe00001ca11b8
>LOG mask = 3ff idx = fff68e95 (295)
*LOG: fffffe000020192a0 LOGSTART: fffffe00002014000 *LOGEND: fffffe0000201bfe0
# 20
[ 000] fffffe000020192a0 xSt0 fffffe00001ca1b88 0000000000000006
0000000000000001
[ 001] fffffe000020192c0 xnd8 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
[ 002] fffffe000020192e0 xnd0 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
//
// USBPORT_Xdpc_End() - USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of
4

[ 003] fffffe00002019300 gNX0 0000000000000000 0000000000000000
fffffe00001ca1b88
[ 004] fffffe00002019320 xbg1 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
[ 005] fffffe00002019340 xbg0 fffffe00001ca1b88 fffffe00001ca1050
fffffe00001ca22e8
//
// USBPORT_Xdpc_iBegin() - USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of
4

[ 006] fffffe00002019360 tmo4 0000000000000000 0000000000000040
fffffe00001ca1c20
[ 007] fffffe00002019380 tmo3 0000000000000000 0000000000989680
fffffe00001ca1c20
[ 008] fffffe000020193a0 tmo2 0000000000000000 0000000000003e8
fffffe00001ca1c20
[ 009] fffffe000020193c0 tmo1 0000000000000000 00000000002625a
fffffe00001ca1c20
[ 010] fffffe000020193e0 tmo0 0000000000003e8 fffffe00001ca1b88
fffffe00001ca1c20
[ 011] fffffe00002019400 hci0 0000000000000000 0000000000000000
0000000000000000
[ 012] fffffe00002019420 xSt0 fffffe00001ca1b88 0000000000000008
0000000000000003
[ 013] fffffe00002019440 xdw4 fffffe00001ca1b88 0000000000000000
0000000000000000
[ 014] fffffe00002019460 xdw2 fffffe00001ca1b88 fffffe00001ca1050
0000000000000002
[ 015] fffffe00002019480 xdB0 fffffe00001ca1b88 fffffe00001ca1050
0000000000000000
//
// USBPORT_Xdpc_Worker_HcIntDpc() DPC/XDPC: 2 of 4

[ 016] fffffe000020194a0 iDP- 0000000000000000 000000000b73e26
```

```
0000000000000000  
//  
// USBPORT_IsrDpc() - Exit()  
  
[ 017] fffffe000020194c0 xSt0 fffffe00001ca1b88 0000000000000007  
0000000000000002  
[ 018] fffffe000020194e0 Xsi1 fffffe00001ca1b88 0000000000000000  
0000000000000000  
//  
// USBPORT_Xdpc_iSignal()- USBPORT_Core_UsbHcIntDpc_Worker() DPC/XDPC: 2 of  
4  
  
[ 019] fffffe00002019500 chgZ 0000000000000000 0000000000b73e26  
0000000000000000
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdpnp

Article • 10/25/2023

The **!usbkd.usbhcdpnp** command displays the Plug and Play (PnP) state history for a USB host controller or root hub.

```
dbgcmd
```

```
!usbkd.usbhcdpnp DeviceExtension
```

Parameters

DeviceExtension

Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of USB host controller. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree

UHCI MINIPORT(s) dt usbport!_USBPORT_MINIPORT_DRIVER fffffe0000090c3d0
...
4)!uhci_info fffffe00001c8f1a0 !devobj fffffe00001c8f050 PCI: VendorId 8086
DeviceId 2938 RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command `!uhci_info fffffe00001c8f1a0`.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdpnp fffffe00001c8f1a0

## PNP STATE LOG (latest at bottom)

##      EVENT          STATE          NEXT
[01] EvFDO_IRP_MN_START_DEVICE    PnpNotStarted    PnpStarted
[02] EvFDO_IRP_MN_QBR_RH        PnpStarted       PnpStarted
```

Here is one way to find the address of the device extension for the PDO of a root hub.

First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
4)!uhci_info fffffe00001c8f1a0 !devobj fffffe00001c8f050 PCI: VendorId 8086
DeviceId 2938 RevisionId 0002
RootHub !hub2_info fffffe00000d941a0 !devstack fffffe00000d94050
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command **!devstack fffffe00000d94050**. Use the [!devstack](#) command to find the address of the PDO and the PDO device extension.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe00000d94050
!DevObj          !DrvObj          !DevExt          ObjectName
> fffffe00000d94050  \Driver\usbhub    fffffe00000d941a0  0000006b
fffffe00000ed4050  \Driver\usbuhci   fffffe00000ed41a0  USBPDO-2
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `fffffe00000ed41a0`.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdpnp fffffe00000ed41a0

## PNP STATE LOG (latest at bottom)

##      EVENT          STATE          NEXT
[01] EvPDO_IRP_MN_START_DEVICE    PnpNotStarted    PnpStarted
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhcdpow

Article • 10/25/2023

The **!usbkd.usbhcdpow** command displays the power state history for a USB host controller or root hub.

```
dbgcmd
```

```
!usbkd.usbhcdpow DeviceExtension
```

Parameters

DeviceExtension

Address of one of the following:

- The device extension for the functional device object (FDO) of a USB host controller.
- The device extension for the physical device object (PDO) a USB root hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of an EHCI host controller. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the [DML](#) command `!ehci_info fffffe00001ca11a0`.

Now pass the address of the device extension to the **!usbhcdpow** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhcdpow fffffe00001ca11a0

dt USBPORT!_FDO_EXTENSION fffffe00001ca15a0

## State History (latest at bottom)

##          EVENT                      STATE
NEXT

[00] FdoPwrEv_D0_DoSetD0_2           FdoPwr_D0_WaitWorker2
FdoPwr_D0_WaitSyncUsb2             dt:0 ms
[01] FdoPwrEv_SyncUsb2_DoChirp    FdoPwr_D0_WaitSyncUsb2
FdoPwr_D0_WaitSyncUsb2             dt:0 ms
[02] FdoPwrEv_Rh_SetPowerSys     FdoPwr_D0_WaitSyncUsb2
FdoPwr_D0_WaitSyncUsb2             dt:0 ms
[03] FdoPwrEv_Rh_SetD0          FdoPwr_D0_WaitSyncUsb2
FdoPwr_D0_WaitSyncUsb2             dt:0 ms
[04] FdoPwrEv_SyncUsb2_Complete   FdoPwr_D0_WaitSyncUsb2
FdoPwr_WaitSx                     dt:50 ms
[05] FdoPwrEv_Rh_Wake            FdoPwr_WaitSx
FdoPwr_WaitSx                     dt:3412 ms
[06] FdoPwrEv_Rh_Wake            FdoPwr_WaitSx
FdoPwr_WaitSx                     dt:283872 ms
[07] FdoPwrEv_Rh_Wake            FdoPwr_WaitSx
FdoPwr_WaitSx                     dt:25481267 ms
```

Here is one way to find the address of the device extension for the PDO of a root hub.
First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
...
```

In the preceding output, you can see the address of the FDO of the root hub displayed as the argument to the command [!devstack fffffe00002320050](#). Use the [!devstack](#) command to find the address of the PDO and the PDO device extension.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe00002320050
!DevObj          !DrvObj          !DevExt          ObjectName
> fffffe00002320050  \Driver\usbhub  fffffe000023201a0  00000002d
```

```
fffffe0000213c050 \Driver\usbehci     fffffe0000213c1a0 USBPDO-3  
...
```

In the preceding output, you can see that the address of the device extension for the PDO of the root hub is `fffffe0000213c1a0`.

Now pass the address of the device extension to the `!usbhcdpow` command.

```
dbgcmd  
  
0: kd> !usbkd.usbhcdpow fffffe0000213c1a0  
  
dt USBPORT!_FDO_EXTENSION fffffe0000213c5a0  
  
## State History (latest at bottom)  
  
##           EVENT                      STATE  
NEXT  
  
...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.hub2_info_from_fdo

Article • 10/25/2023

The **!usbkd.hub2_info_from_fdo** command displays information about a USB hub.

```
dbgcmd
```

```
!usbkd.hub2_info_from_fdo FDO
```

Parameters

FDO

Address of the functional device object (FDO) for a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the FDO for a USB hub. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
```

In the preceding output, the address of the FDO for the hub appears as the argument of the suggested command **!devstack fffffe00002320050**.

Now pass the address of the FDO to the **!hub2_info_from_fdo** command.

```
dbgcmd
```

```
0: kd> !usbkd.hub2_info_from_fdo fffffe00002320050
usbhubext
*****
*
```

```
FDO fffffe00002320050 PDO fffffe0000213c050 HubNumber# 3
dt USBHUB!_DEVICE_EXTENSION_HUB fffffe000023201a0
!usbhublog fffffe000023201a0
RemoveLock fffffe00002320668
FdoFlags fffffe00002320ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)

ObjReferenceList: !usblist fffffe00002320b70, RL
ExceptionList: !usblist fffffe00002321498, EL [Empty]
DmTimerListHead: !usblist fffffe00002321040, TL [Empty]
PdoRemovedListHead: !usblist fffffe00002321478, PL [Empty]
PdoPresentListHead: !usblist fffffe00002321468, PL
WorkItemListHead: !usblist fffffe00002320c80, WI [Empty]
SshBusyListHead: !usblist fffffe00002320dc0, BL

## PnP FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_DEVICE_RELATIONS
...
## POWER FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_POWER - PowerSystemHibernate
...
## HARD RESET STATE HISTORY (latest at bottom)

## EVENT STATE
NEXT

01. HRE_Pause HReset_WaitReady
HReset_Paused
...
## PNP STATE HISTORY (latest at bottom)

## EVENT STATE
NEXT

01. Ev_SYSTEM_POWER FDO_WaitPnpStop
FDO_WaitPnpStop
...
## POWER STATE HISTORY (latest at bottom)

## EVENT STATE
NEXT

01. Ev_SET_POWER_S0 FdoSx_Dx
FdowaitS0IoComplete_Dx
...
```

```
## BUS STATE HISTORY (latest at bottom)

##      EVENT          STATE
NEXT

01. BE_BusSuspend          BS_BusPause
BS_BusSuspend
...
SSH_EnabledStatus: [SSH_ENABLED_VIA_POWER_POLICY]

## SSH STATE HISTORY (latest at bottom)

##      EVENT          STATE
NEXT

01. SSH_Event_ResumeHubComplete    SSH_State_HubPendingResume
SSH_State_HubActive
...
## PORT DATA

PortData 1: !port2_info fffffe000021bf000 Port State = PS_WAIT_CONNECT
PortChangeLock: 0, Pcq_State: Pcq_Run_Idle
    PDO 0000000000000000
...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhuberr

Article • 10/25/2023

The **!usbkd.usbhuberr** command displays a USB hub error record.

```
dbgcmd
```

```
!usbkd.usbhuberr StructAddr
```

Parameters

StructAddr

Address of a **usbhub!_HUB_EXCEPTION_RECORD** structure.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbhub!_HUB_EXCEPTION_RECORD**. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
    ...

```

In the preceding output, you can see the suggested command **!devstack fffffe00002320050**. Enter this command.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe000011f7050

    !DevObj          !DrvObj          !DevExt          ObjectName
> fffffe000011f7050  \Driver\usbhub  fffffe000011f71a0  0000006f
```

```
fffffe00000a21050 \Driver\usbehci     fffffe00000a211a0 USBPDO-8  
...
```

In the preceding output, `fffffe000011f71a0` is the address of the device extension for the functional device object (FDO) of the hub. Pass the address of the device extension to [!usbkd.usbhubext](#).

```
dbgcmd  
  
0: kd> !usbkd.usbhubext fffffe000011f71a0  
  
FDO fffffe000011f7050 PDO fffffe00000a21050 HubNumber# 7  
dt USBHUB!_DEVICE_EXTENSION_HUB fffffe000011f71a0  
!usbhublog fffffe000011f71a0  
RemoveLock fffffe000011f7668  
FdoFlags fffffe000011f7ba0  
  
CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)  
  
ObjReferenceList: !usblast fffffe000011f7b70, RL  
ExceptionList: !usblast fffffe000011f8498, EL [Empty]  
...
```

In the preceding output, `fffffe000011f8498` is the address of the exception list. If the exception list is not empty, it will contain addresses of `_HUB_EXCEPTION_RECORD` structures.

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubext

Article • 10/25/2023

The **!usbkd.usbhubext** command displays information about a USB hub.

```
dbgcmd
```

```
!usbkd.usbhubext DeviceExtension
```

Parameters

DeviceExtension

Address of one of the following:

- The device extension for the functional device object (FDO) of a USB hub.
- The device extension for the physical device object (PDO) of a device that is connected to a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of USB hub.

First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2:!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
```

In the preceding output, you can see the suggested command **!devstack fffffe00002320050**. Enter this command.

```
dbgcmd
```

```

0: kd> !kdexts.devstack fffffe00002320050

!DevObj          !DrvObj          !DevExt          ObjectName
> fffffe00002320050  \Driver\usbhub      fffffe000023201a0  0000002d
    fffffe0000213c050  \Driver\usbehci     fffffe0000213c1a0  USBPDO-3
...

```

In the preceding output, you can see that the address of the device extension for the FDO of the hub is `fffffe000023201a0`.

Now pass the address of the device extension to the `!usbkd.usbhubext` command.

```

dbgcmd

0: kd> !usbkd.usbhubext fffffe000023201a0

FDO fffffe00002320050 PDO fffffe0000213c050 HubNumber# 3
dt USBHUB!_DEVICE_EXTENSION_HUB fffffe000023201a0
!usbhublog fffffe000023201a0
RemoveLock fffffe00002320668
FdoFlags fffffe00002320ba0

CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)

ObjReferenceList: !usblist fffffe00002320b70, RL
ExceptionList: !usblist fffffe00002321498, EL [Empty]
DmTimerListHead: !usblist fffffe00002321040, TL [Empty]
PdoRemovedListHead: !usblist fffffe00002321478, PL [Empty]
PdoPresentListHead: !usblist fffffe00002321468, PL
WorkItemListHead: !usblist fffffe00002320c80, WI [Empty]
SshBusyListHead: !usblist fffffe00002320dc0, BL

## PnP FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_DEVICE_RELATIONS
...
## POWER FUNC HISTORY (latest at bottom)

01. IRP_MN_QUERY_POWER - PowerSystemHibernate
...
## HARD RESET STATE HISTORY (latest at bottom)

##      EVENT                      STATE
NEXT

01. HRE_Pause                  HReset_WaitReady
HReset_Paused
...
## PNP STATE HISTORY (latest at bottom)

```

```

##      EVENT          STATE
NEXT

01. Ev_SYSTEM_POWER          FDO_WaitPnpStop
FDO_WaitPnpStop
...
## POWER STATE HISTORY (latest at bottom)

##      EVENT          STATE
NEXT

01. Ev_SET_POWER_S0          FdoSx_Dx
FdoWaitS0IoComplete_Dx
...
## BUS STATE HISTORY (latest at bottom)

##      EVENT          STATE
NEXT

01. BE_BusSuspend            BS_BusPause
BS_BusSuspend
...
SSH_EnabledStatus: [SSH_ENABLED_VIA_POWER_POLICY]

## SSH STATE HISTORY (latest at bottom)

##      EVENT          STATE
NEXT

01. SSH_Event_ResumeHubComplete    SSH_State_HubPendingResume
SSH_State_HubActive
...
## PORT DATA

PortData 1: !port2_info fffffe000021bf000 Port State = PS_WAIT_CONNECT
PortChangeLock: 0, Pcq_State: Pcq_Run_Idle
    PDO 0000000000000000
...

```

Here is one way to find the address of the device extension for the PDO of a device that is connected to a USB hub. First enter [!usbkd.usb2tree](#).

```

dbgcmd

0: kd> !usbkd.usb2tree
...
2) !ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002

```

```
RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
    Port 1: !port2_info fffffe000021bf000
    Port 2: !port2_info fffffe000021fbfb40
    Port 3: !port2_info fffffe000021c0680 !devstack fffffe00007c882a0
```

In the preceding output, you can see suggested command **!devstack fffffe00007c882a0**. Enter this command.

```
dbgcmd

0: kd> !kdexts.devstack fffffe00007c882a0

    !DevObj          !DrvObj          !DevExt          ObjectName
    fffffe00006ce2260  \Driver\USBSTOR   fffffe00006ce23b0  00000070
> fffffe00007c882a0  \Driver\usbhub   fffffe00007c883f0  USBPDO-4
    ...

```

In the preceding output, you can see that the address of the device extension for the PDO of the device is **fffffe00007c883f0**.

Now pass the address of the device extension to the **!usbhcdpnp** command.

```
dbgcmd

0: kd> !usbkd.usbhubext fffffe00007c883f0

dt USBHUB!_DEVICE_EXTENSION_PDO fffffe00007c883f0
PARENT HUB: FDO fffffe00002320050 !hub2_info fffffe000023201a0
!usbhubinfo fffffe00002320050
PORT NUMBER : 3
IoList: !usblist fffffe00007c888b0, IO
LatchList: !usblist fffffe00007c888e0, LA

## PnP ID's

DeviceId:USB\VID_0781&PID_5530
HardwareId:USB\VID_0781&PID_5530&REV_0100USB\VID_0781&PID_5530
CompatibleId:USB\Class_08&SubClass_06&Prot_50USB\Class_08&SubClass_06USB\Class_08
SerialNumberId:20052444100A47F319CB
UniqueId:3
ProductId:Cruzer

## Pnp Func History (latest at bottom)

01. IRP_MN_QUERY_BUS_INFORMATION
...
## Power Func History (latest at bottom)
```

```

## PNP STATE HISTORY (latest at bottom)

##      EVENT                      STATE
NEXT

01. (6)                          (0)
PDO_WaitPnpStart
02. Ev_PDO_IRP_MN_START          PDO_WaitPnpStart
PDO_WaitPnpStop

## POWER STATE HISTORY (latest at bottom)

##      EVENT                      STATE
NEXT

[EMPTY]

## HARDWARE STATE HISTORY (latest at bottom)

##      EVENT                      STATE
NEXT

01. PdoEv_CreatePdo              (0)
Pdo_Created
02. PdoEv_RegisterPdo           Pdo_Created
Pdo_HwPresent
03. PdoEv_QBR                   Pdo_HwPresent
Pdo_PnpRefHwPresent

## IDLE STATE HISTORY (latest at bottom)

##      EVENT                      STATE
NEXT

[EMPTY]

```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubinfo

Article • 10/25/2023

The **!usbkd.hubinfo** command displays information about a USB hub.

```
dbgcmd
```

```
!usbkd.hubinfo FDO
```

Parameters

FDO

Address of the functional device object (FDO) for a USB hub.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the FDO for a USB hub. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
```

In the preceding output, the address of the FDO for the hub appears as the argument of the suggested command **!devstack fffffe00002320050**.

Now pass the address of the FDO to the **!usbhubinfo** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhubinfo fffffe00002320050
    !DevObj fffffe00002320050 !usbhubext fffffe000023201a0
On Host Controller (0x8086, 0x293c)
```

```
Stat_AsyncResumeStartAt: 2437ee39d29bd528
Stat_AsyncResumeCompleteAt: 24413c77d29bd528
Stat_AsyncResume: 0x3c(60) ms
Stat_SyncResumeStartAt: 2437ee39d29bd528
Stat_SyncResumeCompleteAt: 2437ee39d29bd528
Stat_SyncResume: 0x0(0) ms
Trap Regs: Event, Port, Event (fffffe000023204d0)
    Enable: 0 Port: 0 Event 00000000
Hub Number: # 3
Number Of Ports: 4
dt usbhub!_USBHUB_FDO_FLAGS fffffe00002320ba0
>Is Root
>Power Switching:
    No Power Switching
>Overcurrent:
    Global Overcurrent
>PortIndicators:
    No PortIndicators present
>AllowWakeOnConnect:
    DO NOT WakeOnConnect
>CURRENT Hub Wake on Connectstate:
    HWC_DISARM:- do not wake system on connect/disconnect event
>CURRENT Bus Wake state:
    BUS_DISARM:- bus not armed for wake by this hub
>CURRENT Wake Detect state (WW Irp):
    HUB_DISARM:- no ww irp pending (HUB_WAKESTATE_DISARMED)
Milliamps/Port : 500ma
Power caps (0 = not reported)
    PortPower_Registry : 0
    PortPower_DeviceStatus : 500
    PortPower_CfgDescriptor : 500
    PortPower_HubStatus : 500
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhublog

Article • 10/25/2023

The **!usbkd.usbhublog** command displays the debug log for a USB hub.

```
dbgcmd
```

```
!usbkd.usbhublog DeviceExtension[, NumberOfEntries]
```

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a USB hub.

NumberOfEntries

The number of log entries to display. To display the entire log, set this parameter to -1.

DLL

Usbkd.dll

Examples

Here is one way to find the address of the device extension for the FDO of a USB hub.

First enter **!usbkd.usb2tree**.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
    ...
...
```

In the preceding output, you can see the suggested command **!devstack fffffe00002320050**. Enter this command.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe00002320050
```

!DevObj	!DrvObj	!DevExt	ObjectName
> fffffe00002320050	\Driver\usbhub	fffffe000023201a0	0000002d
fffffe0000213c050	\Driver\usbehci	fffffe0000213c1a0	USBPDO-3
...			

In the preceding output, `fffffe000023201a0` is the address of the device extension for the FDO of the hub.

Now pass the address of the device extension to `!usbhublog`. In this example, the second argument limits the display to 10 log entries.

```
dbgcmd

0: kd> !usbkd.usbhublog fffffe000023201a0, 10

LOG@: fffffe000023201a0 (usbhub!_DEVICE_EXTENSION_HUB)
>LOG mask = ff idx = fffffa333 (33)
*LOG: fffffe00002321ca0 LOGSTART: fffffe00002321640 *LOGEND: fffffe00002323620
# 20
[ 000] fffffe00002321ca0 HDec 0000000000000000 fffffe000002904d0
0000000000000001
[ 001] fffffe00002321cc0 HPCd 0000000000000000 0000000000000002
0000000000000004
[ 002] fffffe00002321ce0 qwk- 0000000000000000 fffffe000021c11c0
0000000000000000
[ 003] fffffe00002321d00 pq-- 0000000000000000 0000000000000002
0000000000000004
[ 004] fffffe00002321d20 _6p4 0000000000000000 0000000000000000
0000000000000004
[ 005] fffffe00002321d40 _6p1 0000000000000000 0000000000000003
0000000000000004
[ 006] fffffe00002321d60 pq++ 0000000000000000 0000000000000003
0000000000000004
[ 007] fffffe00002321d80 pq++ 0000000000000000 0000000000000006
0000000000000004
[ 008] fffffe00002321da0 _6p0 0000000000000000 fffffe000021c11c0
0000000000000004
[ 009] fffffe00002321dc0 pqDP 0000000000000000 fffffe000021c11d8
0000000000000006
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubmddevext

Article • 10/25/2023

The `!usbkd.usbhubmddevext` command displays a `usbhub!_DEVICE_EXTENSION_HUB` structure if one is present in a crash dump that was generated as a result of a [Bug Check 0xFE](#).

```
dbgcmd
```

```
!usbkd.usbhubmddevext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubmdpd

Article • 10/25/2023

The `!usbkd.usbhubmdpd` command displays a `usbhub!_HUB_PORT_DATA` structure if one is present in a crash dump that was generated as a result of [Bug Check 0xFE](#).

dbgcmd

```
!usbkd.usbhubmdpd [PortNum]
```

Parameters

PortNum

A USB port number. If you specify a port number, this command displays the structure (if one is present) that represents the specified port. If you do not specify a port number, this command displays the structure (if one is present) on which [Bug Check 0xFE](#) was initiated.

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubpd

Article • 10/25/2023

The **!usbkd.usbhubpd** command displays information about a USB port.

```
dbgcmd
```

```
!usbkd.usbhubpd StructAddr
```

Parameters

StructAddr

Address of a **usbhub!_HUB_PORT_DATA** structure. To get the addresses of these structures, use [!usbhubext](#).

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbhub!_HUB_PORT_DATA**. First enter **!usbkd.usb2tree**.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
    RootHub !hub2_info fffffe000023201a0 !devstack fffffe00002320050
    ...
...
```

In the preceding output, you can see the suggested command **!devstack fffffe00002320050**. Enter this command.

```
dbgcmd
```

```
0: kd> !kdexts.devstack fffffe00002320050
    !DevObj          !DrvObj          !DevExt          ObjectName
> fffffe00002320050  \Driver\usbhub   fffffe000023201a0  0000002d
```

```
fffffe0000213c050 \Driver\usbehci     fffffe0000213c1a0 USBPDO-3  
...
```

In the preceding output, you can see that the address of the device extension for the FDO of the hub is `fffffe000023201a0`.

Pass the address of the device extension to the `!usbhubext` command.

```
dbgcmd  
  
0: kd> !usbkd.usbhubext fffffe000023201a0  
  
FDO fffffe00002320050 PDO fffffe0000213c050 HubNumber# 3  
dt USBHUB!_DEVICE_EXTENSION_HUB fffffe000023201a0  
!usbhublog fffffe000023201a0  
RemoveLock fffffe00002320668  
FdoFlags fffffe00002320ba0  
  
CurrentPowerIrp: System (0000000000000000) Device (0000000000000000)  
...  
## PORT DATA  
  
PortData 1: !port2_info fffffe000021bf000 Port State = PS_WAIT_CONNECT  
PortChangeLock: 0, Pcq_State: Pcq_Run_Idle  
    PDO 0000000000000000  
....
```

In the preceding output, `fffffe000021bf000` is the address of a `_HUB_PORT_DATA` structure. Pass this address to `!usbhubpd`.

```
dbgcmd  
  
0: kd> !usbkd.usbhubpd fffffe000021bf000  
PortNumber: 1  
Parent Hub FDO fffffe00002320050  
Device PDO <NULL>  
dt USBHUB!_HUB_PORT_DATA fffffe000021bf000  
dt USBHUB!_PORTDATA_FLAGS fffffe000021bf968  
  
PortChangelist: !usblist fffffe000021bf1c8, CL [Empty]  
  
## Port Indicators Log (latest at bottom)  
  
## Event           State          Next  
[EMPTY]  
  
## Port Change Queue History (latest at bottom)  
  
## Event           State          Next  
PcqEv_Suspend   PcqEv_Resume  PcqEv_ChDone Tag
```

```
01. PCE_Resume          Pcq_Stop           Pcq_Pause
PcqEv_Reset   PcqEv_Reset   REQUEST_RESUME
...          Pcq_Run_wBusy    Pcq_Run_Idle

## Port Status History (latest at bottom)

##      Current State      Change Event      PDO      CEOSP
H/W Port REG Frame Inserted

01. PS_WAIT_CONNECT     REQUEST_PAUSE    0000000000000000 00000 100
Age:000 512498
...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbhubs

Article • 10/25/2023

The **!usbkd.usbhubs** command displays information about USB hubs.

```
dbgcmd
```

```
!usbkd.usbhubs a[v]
!usbkd.usbhubs x[v]
!usbkd.usbhubs r[v]
```

Parameters

a

Display all hubs.

r

Display root hubs.

x

Display external hubs.

v

The output is verbose. For example, **!usbhubs rv** displays verbose output about all root hubs.

DLL

Usbkd.dll

Examples

Here is an example of verbose output from the **!usbhubs** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbhubs rv

args 'rv'
level: r v
ROOT HUBS
*LIST -- Root Hub List @ fffff8000217f440
```

```

*flink, blink fffffe000023215c0,fffffe000011f85c0
-----

[0] entry @: fffffe000023201a0
    !DevObj fffffe00002320050 !usbhubext fffffe000023201a0
On Host Controller (0x8086, 0x293c)
    Stat_AsyncResumeStartAt: 2437ee39d29bd498
    Stat_AsyncResumeCompleteAt: 24413c77d29bd498
    Stat_AsyncResume: 0x3c(60) ms
    Stat_SyncResumeStartAt: 2437ee39d29bd498
    Stat_SyncResumeCompleteAt: 2437ee39d29bd498
    Stat_SyncResume: 0x0(0) ms
Trap Regs: Event, Port, Event (fffffe000023204d0)
    Enable: 0 Port: 0 Event 00000000
Hub Number: # 3
Number Of Ports: 4
dt usbhub!_USBHUB_FDO_FLAGS fffffe00002320ba0
>Is Root
>Power Switching:
    No Power Switching
>Overcurrent:
    Global Overcurrent
>PortIndicators:
    No PortIndicators present
>AllowWakeOnConnect:
    DO NOT WakeOnConnect
>CURRENT Hub Wake on Connectstate:
    HWC_DISARM:- do not wake system on connect/disconnect event
>CURRENT Bus Wake state:
    BUS_DISARM:- bus not armed for wake by this hub
>CURRENT Wake Detect state (WW Irp):
    HUB_DISARM:- no ww irp pending (HUB_WAKESTATE_DISARMED)
Milliamps/Port : 500ma
Power caps (0 = not reported)
    PortPower_Registry : 0
    PortPower_DeviceStatus : 500
    PortPower_CfgDescriptor : 500
##        PortPower_HubStatus : 500

-----
[1] entry @: fffffe000008b91a0
    !DevObj fffffe000008b9050 !usbhubext fffffe000008b91a0
On Host Controller (0x8086, 0x2937)
...

```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usblist

Article • 10/25/2023

The **!usbkd.usblist** command displays a linked list of structures of a specified type.

dbgcmd

```
!usbkd.usblist ListAddr, ListType
```

Parameters

ListAddr

Address of a linked list of structures. To find addresses of linked lists maintained by the USB port driver, use [!usbhcdext](#). To find addresses of linked list maintained by the USB hub driver, use [!usbhubext](#).

ListType

One of the following list types.

List type	Structure
BC	usbport!_BUS_CONTEXT
EP	usbport!_HCD_ENDPOINT
TT	usbport!_TRANSACTION_TRANSLATOR
DL	usbport!_USBD_DEVICE_HANDLE
PL	usbhub!_DEVICE_EXTENSION_PDO
EL	usbhub!_HUB_EXCEPTION_RECORD
RL	usbhub!_HUB_REFERENCE_LIST_ENTRY
TL	usbhub!_HUB_TIMER_OBJECT
WI	usbhub!_HUB_WORKITEM
IO	usbhub!_IO_LIST_ENTRY
LA	usbhub!_LATCH_LIST_ENTRY
CL	usbhub!_PORT_CHANGE_CONTEXT
BL	usbhub!_SSP_BUSY_HANDLE

DLL

Usbkd.dll

Examples

Here is one way to find the address of a linked list. First enter [!usbkd.usb2tree](#).

```
dbgcmd  
0: kd> !usbkd.usb2tree  
...  
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 ...  
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command [!ehci_info fffffe00001ca11a0](#).

Either click the DML command or pass the address of the device extension to [!usbhcdext](#).

```
dbgcmd  
0: kd> !usbkd.usbhcdext fffffe00001ca11a0  
  
HC Flavor 1000 FDO fffffe00001ca1050  
Root Hub: FDO fffffe00002320050 !hub2_info fffffe000023201a0  
...  
DeviceHandleList: !usblist fffffe00001ca23b8, DL  
...
```

In the preceding output, fffffe00001ca23b8 is the address of a linked list of [usbport!_USBD_DEVICE_HANDLE](#) structures.

Now pass the address of the linked list to [!usblist](#).

```
dbgcmd  
0: kd> !usblist fffffe00001ca23b8, DL  
list: fffffe00001ca23b8 DL  
-----  
!usbdevh fffffe000020f9590  
SSP [IdleReady] (0)  
PCI\VEN_Xxxx Xxxx Corporation  
Root Hub  
DriverName :  
-----
```

```
!usbdevh fffffe00001bce250
SSP [IdleReady] (0)
USB\XXXX XXXX Corporation
Speed: HIGH, Address: 1, PortPathDepth: 1, PortPath: [3 0 0 0 0 0]
DriverName :\Driver\USBSTOR      !devstack fffffe000053ef2a0
-----
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbpo

Article • 10/25/2023

The **!usbkd.usbpo** command displays the internal list of outstanding USB power requests.

```
dbgcmd
```

```
!usbkd.usbpo
```

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbpdos

Article • 10/25/2023

The **!usbkd.usbpdos** command displays information about all physical device objects (PDOs) created by the USB hub driver.

```
dbgcmd
```

```
!usbkd.usbpdos
```

DLL

Usbkd.dll

Examples

Here's an example of the output of the **!usbpdos** command.

```
dbgcmd
```

```
0: kd> !usbkd.usbpdos

ext fffffe00006c513f0
VID 0a12 PID 0001 REV 0309
dt USBHUB!_USB_DEVICE_DESCRIPTOR fffffe00006c51960
Vendor: XXXX
XXXX Corporation
dd Class: (224)(e0)
(224)(e0)
HubFdo: fffffe00000d94050, port: 2
<null>
SystemWake 5 SystemHibernate(S4)
wake_irp_list_head = fffffe00006c51cb0
wake_irp = fffffe00006c51cb0
PDO Times:
    Stat_PdoCreatedAt: 0d29bbd58
    Stat_PdoEnumeratedAt: b65ace75d29bbd58
    Stat: Enumeration Time: 0xa7d3842a(-1479310294) ms
    Stat_Pdo_SetD0_StartAt: 0d29bbd58
    Stat_Pdo_SetD0_CompleteAt: 0d29bbd58
    Stat: PDO Set_D0 time: 0x0(0) ms

##
ext fffffe00007c883f0
VID 0781 PID 5530 REV 0100
```

```
dt USBHUB!_USB_DEVICE_DESCRIPTOR fffffe00007c88960
Vendor: XXXX
XXXX Corporation
dd Class: (0)(0)
(8)Class_UsbStorage
HubFdo: fffffe00002320050, port: 3
:XXXX
SystemWake 5 SystemHibernate(S4)
wake_irp_list_head = fffffe00007c88cb0
wake_irp = fffffe00007c88cb0
PDO Times:
    Stat_PdoCreatedAt: 0d29bbd58
    Stat_PdoEnumeratedAt: 2380af52d29bbd58
    Stat: Enumeration Time: 0x9a24226c(-1708907924) ms
    Stat_Pdo_SetD0_StartAt: 0d29bbd58
    Stat_Pdo_SetD0_CompleteAt: 0d29bbd58
##    Stat: PDO Set_D0 time: 0x0(0) ms
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbpdoxls

Article • 10/25/2023

The **!usbkd.usbpdoxls** command displays information about all physical device objects (PDOs) created by the USB hub driver. For each PDO, this command displays a row of comma-separated values.

```
dbgcmd
```

```
!usbkd.usbpdoxls
```

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbpnp

Article • 10/25/2023

The **!usbkd.usbpnp** command displays state context information about a USB hub.

dbgcmd

!usbkd.usbpnp DeviceExtension

Parameters

DeviceExtension

Address of the device extension for the functional device object (FDO) of a USB hub.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbportisasyncadv

Article • 10/25/2023

The **!usbkd.usbportisasyncadv** command checks all EHCI miniport drivers for an EHCI Interrupt on Async Advance issue.

dbgcmd

!usbkd.usbportisasyncadv

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbportmdportlog

Article • 10/25/2023

The `!usbkd.usbportmdportlog` command displays the USBPORT debug log if it is present in a crash dump that was generated as a result of [Bug Check 0xFE](#).

```
dbgcmd
```

```
!usbkd.usbportmdportlog
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

Examples

Here is an example of a portion of the output of `!usbportmdportlog`.

```
dbgcmd
```

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.usbportmdportlog
Minidump USBPORT DEBUG_LOG buffer size 32768, entries 1024, index 400
*LLOG: 0000000113be9600 LOGSTART: 0000000113be6400 *LOGEND: 0000000113bee3e0
# 1024
[ 000] 0000000113be9600 Bfe2 fffffe0001416802c fffffe000020a44f0
fffffe0001416801c
[ 001] 0000000113be9620 Bfe0 0000000000000000 fffffe000039f4720
fffffe00000b76cb0
[ 002] 0000000113be9640 epr+ fffffe000043ee010 fffffe000008f5b80
fffffe00002820a0c
[ 003] 0000000113be9660 a1TR fffffe00002820a0c fffffe000039f4720
fffffe00000b76cb0
//
// USBPORT_Core_AllocTransferEx()
// transfer: fffffe00002820a0c
```

```
// Urb:                      fffffe000039f4720
// Irp:                       fffffe00000b76cb0

[ 004] 0000000113be9680 TRcs 0000000000000060 0000000000000468
0000000000000001
[ 005] 0000000113be96a0 urbi fffffe0001422c4cc 0000000000000012
000000000000000b
[ 006] 0000000113be96c0 dURB fffffe000039f4720 fffffe00000b76cb0
000000000000000b
//
// USBPORT_ProcessURB() - Device Handle valid and has been referenced
// Urb:                      fffffe000039f4720
// Irp:                       fffffe00000b76cb0
// function:                 URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 007] 0000000113be96e0 vld> fffffe0001421ddd0 0000000000000004
fffffe000039f4720
//
// USBPORT_NeoValidDeviceHandle()
// DeviceHandle:             fffffe0001421ddd0
// ReferenceObj:             fffffe000039f4720

[ 008] 0000000113be9700 devH fffffe0001421ddd0 fffffe000039f4720
0000000000000000
[ 009] 0000000113be9720 pURB fffffe000039f4720 fffffe00000b76cb0
000000000000000b
//
// USBPORT_ProcessURB()
// Urb:                      fffffe000039f4720
// Irp:                       fffffe00000b76cb0
// function:                 URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 010] 0000000113be9740 PURB fffffe000039f4720 fffffe000020a44f0
000000000000000b
//
// USBPORT_ProcessURB() - Exit STATUS_PENDING
// Urb:                      fffffe000039f4720
// Irp:                       fffffe000020a44f0
// function:                 URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

[ 011] 0000000113be9760 Urb< 000000000000103 000000000000000b
0000000000000000
[ 012] 0000000113be9780 xSt0 fffffe000012bbf18 0000000000000006
0000000000000001
[ 013] 0000000113be97a0 xnd8 fffffe000012bbf18 fffffe000012bb050
0000000000000000
[ 014] 0000000113be97c0 xnd0 fffffe000012bbf18 fffffe000012bb050
0000000000000000
//
// USBPORT_Xdpc_End()

[ 015] 0000000113be97e0 mapF 0000000000000000 0000000000000000
0000000000000000
[ 016] 0000000113be9800 map5 0000000000000000 0000000000000000
0000000000000000
```

```
[ 017] 0000000113be9820 DMAx 0000000000000000 0000000000000000  
0000000000000000  
[ 018] 0000000113be9840 subx 0000000000000000 0000000000000000  
ffffe0001416801c  
[ 019] 0000000113be9860 tmoZ 0000000000000000 fffffe0001416801c  
ffffe000141680a4  
...  
...
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbportmddcontext

Article • 10/25/2023

The **!usbkd.usbportmddcontext** command displays USBPORT context data if it is present in a crash dump that was generated as a result of [Bug Check 0xFE](#).

```
dbgcmd
```

```
!usbkd.usbportmddcontext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbportmddevext

Article • 10/25/2023

The **!usbkd.usbportmddevext** command displays a **usbport!_DEVICE_EXTENSION** structure if one is present in a crash dump that was generated as a result [Bug Check 0xFE](#).

```
dbgcmd
```

```
!usbkd.usbportmddevext
```

DLL

Usbkd.dll

Remarks

Use this command only when you are debugging a crash dump file that was generated as a result of [Bug Check 0xFE: BUGCODE_USB_DRIVER](#).

Examples

Here is an example of the output of **!usbportmddevext**.

```
dbgcmd
```

```
1: kd> !analyze -v
*** ...
BUGCODE_USB_DRIVER (fe)
...
1: kd> !usbkd.usbportmddevext
USBPORT.SYS DEVICE_EXTENSION DATA:
Hcd FDO Extension:
Sig:4f444648 HFDO
CurrentPnpFunc: 0x00000008
PnpFuncHistoryIdx: 0x0000000d
CurrentPowerFunc: 0x00000000
PowerFuncHistoryIdx: 0x00000000
PnpLogIdx: 0x00000002
IoRequestCount: 0x00000007
IoRequestAsyncCallbackCount: 0xffffffff
IoRequestAllow: 0x00000000
Pnp Func History (idx 13)
...
```

```

[02] pnp 13 (0d) IRP_MN_FILTER_RESOURCE_REQUIREMENTS
[...
Power Func History (idx 0)
[01] pnp 255 (ff) ??? (x0) PowerDeviceUnspecified
...
    **Power and Wake -----
    selective suspend:on (1)
    PowerFlags (00000080):
*---FDO---*
PMDebug: 0x00000000
MinAllocedBw: 0x00000000
MaxAllocedBw: 0x00000000
## ...

## XDPC HISTORY_UsbHcIntDpc

State History (idx 2)
EVENT, STATE, NEXT
Log[3] @ 000000d9e7c615cc
Ev_Xdpc_Worker      XDPC_DpcQueued      XDPC_Running
## ...

## XDPC HISTORY_UsbDoneDpc

State History (idx 0)
EVENT, STATE, NEXT
Log[1] @ 000000d9e7c61774
Ev_Xdpc_Worker      XDPC_DpcQueued      XDPC_Running
## ...

## XDPC HISTORY_UsbMapDpc

State History (idx 3)
EVENT, STATE, NEXT
Log[4] @ 000000d9e7c6196c
## ...

## XDPC HISTORY_UsbIocDpc

State History (idx 0)
EVENT, STATE, NEXT
Log[1] @ 000000d9e7c61b04
Ev_Xdpc_Worker      XDPC_DpcQueued      XDPC_Running
...

```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbtriage

Article • 10/25/2023

The **!usbkd.usbtriage** command displays a list of USB drivers and device objects.

```
dbgcmd
```

```
!usbkd.usbtriage
```

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbtt

Article • 10/25/2023

The **!usbkd.usbtt** command displays information from a **USBPORT!_TRANSACTION_TRANSLATOR** structure.

```
dbgcmd
```

```
!usbkd.usbtt StructAddr
```

Parameters

StructAddr

Address of a **usbport!_TRANSACTION_TRANSLATOR** structure. To get the transaction translator list for a USB host controller, use the **!usbkd.usbhcdext** command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_TRANSACTION_TRANSLATOR** structure. First enter **!usbkd.usb2tree**.

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
2)!ehci_info fffffe00001ca11a0 !devobj fffffe00001ca1050 PCI: VendorId 8086
DeviceId 293c RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the **DML** command **!ehci_info fffffe00001ca11a0**.

Either click the DML command or pass the address of the device extension to **!usbhcdext** to get the address of **GlobalTtListHead**. Pass that address to **!usbkd.usblist**, which will display addresses of **_TRANSACTION_TRANSLATOR** structures.

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbttx

Article • 10/25/2023

The **!usbkd.usbttx** command displays information from a **usbport!_HCD_TRANSFER_CONTEXT** structure.

```
dbgcmd
```

```
!usbkd.usbttx StructAddr
```

Parameters

StructAddr

Address of a **usbport!_HCD_TRANSFER_CONTEXT** structure. To get the transfer list for a USB host controller, use the [!usbkd.usbhcdext](#) command.

DLL

Usbkd.dll

Examples

Here is one way to find the address of a **usbport!_HCD_TRANSFER_CONTEXT** structure. First enter [!usbkd.usb2tree](#).

```
dbgcmd
```

```
0: kd> !usbkd.usb2tree
...
4)!uhci_info fffffe00001c8f1a0 !devobj fffffe00001c8f050 PCI: VendorId 8086
DeviceId 2938 RevisionId 0002
...
```

In the preceding output, the address of the device extension of the FDO is displayed as the argument of the DML command [!uhci_info fffffe00001c8f1a0](#).

Either click the DML command or pass the address of the device extension to [!usbhcdext](#) to get the transfer list.

```
dbgcmd
```

```

0: kd> !usbkd.usbhcdext fffffe00001c8f1a0
...
## I/O TRANSFER LIST(s)

1.) Transfer Request Priority List: (TxQueued) Type: 0-NotSplit, 1-Parent,
2-Child
-----
-----
[000]!usbttx fffffe0000653401c !usbep fffffe00004730c60 !irp
fffffe00004221220 State: (7)TX_Mapped_inMp
    Priority: 0, Type: 0, Flags= 0000000a, SequenceNum: 10, SplitIdx: 0
    InLen: 4096, OutLen: 0 Status: USBD_STATUS_PENDING (0x40000000)
...

```

In the preceding output, `fffffe0000653401c` is the address of an `_HCD_TRANSFER_CONTEXT` structure. Pass this address to `!usbttx`.

```

dbgcmd

0: kd> !usbkd.usbttx fffffe0000653401c

dt usbport!_HCD_TRANSFER_CONTEXT fffffe0000653401c
dt usbport!_TRANSFER_PARAMETERS fffffe0000653417c

## TX HISTORY

## EVENT, STATE, NEXT (latest at bottom)

[01] (23)Ev_TX_Icsq, (0)TX_Undefined, (1)TX_InQueue
[02] (5)Ev_TX_MapTransfer, (1)TX_InQueue, (2)TX_MapPending
[03] (7)Ev_TX_MpSubmitSuccess, (2)TX_MapPending, (7)TX_Mapped_inMp

**DMA**
dt usbport!_TRANSFER_SG_LIST fffffe0000653439c
SgCount: 1 MdlVirtualAddress: fffffe0000437000 MdlSystemAddress:
fffffe0000437000
[0] dt usbport!_TRANSFER_SG_ENTRY fffffe000065343bc
: sysaddr: 0000000000000000 len 0x00001000(4096) offset 0x00000000(0)
phys 00000000'ded90000
---
dt usbport!_SCATTER_GATHER_ENTRY fffffe000065343ec
dt _SCATTER_GATHER_LIST fffffe00001bc231c
NumberOfElements = 1
[0] dt _SCATTER_GATHER_ELEMENT fffffe00001bc232c
:phys 00000000'ded90000 len 0x00001000(4096)
```

See also

[USB 2.0 Debugger Extensions](#)

Universal Serial Bus (USB) Drivers

!usbkd.usbusb2ep

Article • 10/25/2023

The **!usbkd.usbusb2ep** command displays information from a **usbport!_USB2_EP** structure.

dbgcmd

```
!usbkd.usbusb2ep StructAddr
```

Parameters

StructAddr

Address of a **usbport!_USB2_EP** structure. To get the address of **usbport!_USB2_EP** structure, use [**!usbkd.usb2**](#).

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbusb2tt

Article • 10/25/2023

The **!usbkd.usbusb2tt** command displays information from a **usbport!_TT** structure.

dbgcmd

```
!usbkd.usbusb2tt StructAddr
```

Parameters

StructAddr

Address of a **usbport!_TT** structure.

DLL

Usbkd.dll

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

!usbkd.usbver

Article • 10/25/2023

The **!usbkd.usbver** command displays the USBD interface version of the USB driver stack.

```
dbgcmd
```

```
!usbkd.usbver
```

DLL

Usbkd.dll

Remarks

The value of the USBD interface version is stored in the variable `usbport!usbd_version`.

Examples

Here is an example of the output of **!usbkd.usbver**.

```
dbgcmd
```

```
1: kd> !usbkd.usbver
```

```
USBD VER 600 USB stack is VISTA
```

See also

[USB 2.0 Debugger Extensions](#)

[Universal Serial Bus \(USB\) Drivers](#)

[USBD_IsInterfaceVersionSupported](#)

RCDRKD Extensions

Article • 10/25/2023

This section describes the RCDRKD debugger extension commands. These commands display WPP trace messages created by drivers. Starting with Windows 8, you no longer need a separate trace message format (TMF) file to parse WPP messages. The TMF information is stored in the regular symbol file (PDB file).

Starting in Windows 10, kernel-mode and user-mode drivers can use [Inflight Trace Recorder \(IFR\) for logging traces](#). Your kernel-mode driver can use the RCDRKD commands to read messages from the circular buffers, format the messages, and display the messages in the debugger.

Note You cannot use the RCDRKD commands to view UMDF driver logs, UMDF framework logs, and KMDF framework logs. To view those logs, use [Windows Driver Framework Extensions \(Wdfkd.dll\)](#) commands.

The RCDRKD debugger extension commands are implemented in Rcdrkd.dll. To load the RCDRKD commands, enter `.load rcdrkd.dll` in the debugger.

The following two commands are the primary commands for displaying trace messages.

- [`!rcdrkd.rcdrlogdump`](#)
- [`!rcdrkd.rcdrcrashdump`](#)

The following auxiliary commands provide services related to displaying and saving trace messages.

- [`!rcdrkd.rcdrloglist`](#)
- [`!rcdrkd.rcdrlogsave`](#)
- [`!rcdrkd.rcdrsearchpath`](#)
- [`!rcdrkd.rcdrsettraceprefix`](#)
- [`!rcdrkd.rcdrtmffile`](#)
- [`!rcdrkd.rcdrtraceprtdebug`](#)

The [`!rcdrkd.rcdrhelp`](#) displays help for the RCDRKD commands in the debugger.

See also

[WPP Software Tracing](#)

[Using the Framework's Event Logger](#)

USB 3.0 Extensions

!rcdrkd.rcdrhelp

Article • 10/25/2023

The **!rcdrkd.rcdrhelp** command displays help for the RCDRKD debugger extension commands.

DLL

Rcdrkd.dll

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrcrashdump

Article • 10/25/2023

The [!rcdrkd.rcdrcrashdump](#) extension is used with a minidump file to display the recorder log (if the log is present in the minidump).

dbgcmd

```
!rcdrkd.rcdrcrashdump TraceProviderGuid  
!rcdrkd.rcdrcrashdump DriverName
```

Parameters

TraceProviderGuid

GUID of the trace provider. This parameter must include braces: {*guid*}

DriverName

The name of the driver. The driver name can be used instead of the trace provider GUID for drivers that use the Inflight Trace Recorder (IFR).

DLL

Rcdrkd.dll

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrlogdump

Article • 10/25/2023

The `!rcdrkd.rcdrlogdump` extension displays the trace messages from all recorder buffers of a driver or set of drivers.

dbgcmd

```
!rcdrkd.rcdrlogdump DriverName [DriverName ...]  
!rcdrkd.rcdrlogdump DriverName -a Address
```

Parameters

DriverName

The name of a driver, not including the .sys extension.

Address

If *Address* is specified, this command displays the trace messages from the log buffer at the specified address.

DLL

Rcdrkd.dll

Examples

The following example shows a portion of the output of the `!rcdrlogdump` command.

dbgcmd

```
3: kd> !rcdrlogdump usbhci.sys  
Trace searchpath is:  
  
Trace format prefix is: %7!u!: %!FUNC! -  
Log dump command          Log ID          Size  
=====  =====  =====  
!rcdrlogdump  usbhci -a ffffffa8005ff2b60  03 SLT02 DCI04  1024  
!rcdrlogdump  usbhci -a ffffffa8005ff2010  03 SLT02 DCI03  1024  
!rcdrlogdump  usbhci -a ffffffa8005b36010  03 SLT01 DCI03  1024  
!rcdrlogdump  usbhci -a ffffffa8005b379e0  03 SLT01 DCI04  1024  
!rcdrlogdump  usbhci -a ffffffa8005b33350  03 SLT02 DCI01  1024  
!rcdrlogdump  usbhci -a ffffffa8005b2bb60  03 SLT01 DCI01  1024  
!rcdrlogdump  usbhci -a ffffffa8005a2bb60  03 CMD
```

```
!rcdrlogdump usbhci -a ffffffa8005a1ab60 03 INT00          1024
!rcdrlogdump usbhci -a ffffffa8005085330 03 RUNDOWN        512
!rcdrlogdump usbhci -a ffffffa8005311780 03 1033 0194        1024
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbhci
.pdb
--- start of log ---
03 RUNDOWN      6: Controller_RetrievePciData - PCI Bus.Device.Function:
48.0.0
03 RUNDOWN      7: Controller_RetrievePciData - PCI: VendorId 0x1033 DeviceId
0x0194 RevisionId 0x03
03 RUNDOWN      8: Controller_QueryDeviceFlagsFromKse - Found DeviceFlags
0x101800 for USBXHCI:PCI\VEN_1033&DEV_0194
03 RUNDOWN      9: Controller_RetrieveDeviceFlags - DeviceFlags 0x101804
...
03 SLT01 DCI03 89911: TransferRing_DispatchEventsAndReleaseLock - 1.3.0:
Mapping Complete : Path 1 TransferRingState_Idle Events 0x00000000
03 SLT01 DCI04 89912: TransferRing_DispatchEventsAndReleaseLock - 1.4.0:
Mapping Begin : Path 3 TransferRingState_Mapping Events 0x00000000
03 SLT01 DCI04 89913: TransferRing_DispatchEventsAndReleaseLock - 1.4.0:
Mapping Complete : Path 3 TransferRingState_Idle Events 0x00000000
---- end of log ----
```

The preceding output contains messages from several log buffers. To see messages from a single log buffer, use the **-a** parameter, and specify the address of the log buffer. The following example shows how to display the messages from the log buffer at address fffffa8005ff2b60.

```
dbgcmd

3: kd> !rcdrlogdump usbhci -a ffffffa8005ff2b60
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbhci
.pdb
--- start of log ---
70914: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete :
Path 1 TransferRingState_Idle Events 0x00000000
70916: TransferRing_TransferEventHandler - 2.4.0: TransferEventTrb
0xFFFFFA8005A3FBF0 CC_SUCCESS Length 31 EventData 1 Pointer
0xfffffa8005b96210
70917: TransferRing_TransferEventHandler - 2.4.0: WdfRequest
0x0000057FFA469FD8 transferData 0xFFFFFA8005B961B0
70918: TransferRing_TransferComplete - 2.4.0: WdfRequest 0x0000057FFA469FD8
completed with STATUS_SUCCESS BytesProcessed 31
70922: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin :
Path 3 TransferRingState_Mapping Events 0x00000000
70923: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete :
Path 3 TransferRingState_Idle Events 0x00000000
81064: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin :
```

```
Path 1 TransferRingState_Mapping Events 0x00000000
81065: TransferRing_StageRetrieveFromRequest - 2.4.0: WdfRequest
0x0000057FFA469FD8 TransferData 0xFFFFFA8005B961B0 Function 0x9 Length 31
TransferSize 31 BytesProcessed 0
81066: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete :
Path 1 TransferRingState_Idle Events 0x00000000
81068: TransferRing_TransferEventHandler - 2.4.0: TransferEventTrb
0xFFFFFA8005A40270 CC_SUCCESS Length 31 EventData 1 Pointer
0xfffffa8005b96210
81069: TransferRing_TransferEventHandler - 2.4.0: WdfRequest
0x0000057FFA469FD8 transferData 0xFFFFFA8005B961B0
81070: TransferRing_TransferComplete - 2.4.0: WdfRequest 0x0000057FFA469FD8
completed with STATUS_SUCCESS BytesProcessed 31
81074: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Begin :
Path 3 TransferRingState_Mapping Events 0x00000000
81075: TransferRing_DispatchEventsAndReleaseLock - 2.4.0: Mapping Complete :
Path 3 TransferRingState_Idle Events 0x00000000
---- end of log ----
```

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrloglist

Article • 10/25/2023

The **!rcdrkd.rcdrloglist** extension displays a list of the recorder logs owned by a driver or a set of drivers.

dbgcmd

```
!rcdrkd.rcdrloglist DriverName [DriverName ...]
```

Parameters

DriverName

The name of a driver, not including the .sys extension.

DLL

Rcdrkd.dll

Remarks

This command is relevant only for drivers that log messages to different logs by using the WppRecorder API.

Examples

The following example displays a list of all recorder logs owned by the USB 3.0 host controller driver (usbxhci.sys).

dbgcmd

```
3: kd> !rcdrloglist usbxhci
Log dump command
=====
!rcdrlogdump usbxhci -a ffffffa8005ff2b60 03 SLT02 DCI04      1024
!rcdrlogdump usbxhci -a ffffffa8005ff2010 03 SLT02 DCI03      1024
!rcdrlogdump usbxhci -a ffffffa8005b36010 03 SLT01 DCI03      1024
!rcdrlogdump usbxhci -a ffffffa8005b379e0 03 SLT01 DCI04      1024
!rcdrlogdump usbxhci -a ffffffa8005b33350 03 SLT02 DCI01      1024
!rcdrlogdump usbxhci -a ffffffa8005b2bb60 03 SLT01 DCI01      1024
!rcdrlogdump usbxhci -a ffffffa8005a2bb60 03 CMD            1024
```

```
!rcdrlogdump usbhci -a ffffffa8005a1ab60 03 INT00          1024
!rcdrlogdump usbhci -a ffffffa8005085330 03 RUNDOWN        512
!rcdrlogdump usbhci -a ffffffa8005311780 03 1033 0194      1024
```

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrlogsave

Article • 10/25/2023

The **!rcdrkd.rcdrlogsave** extension saves the recorder buffers of a driver.

dbgcmd

```
!rcdrkd.rcdrlogsave DriverName [CaptureFilename ]
```

Parameters

DriverName

The name of the driver, not including the .sys extension.

CaptureFileName

The name of the file (not including the .etl extension) in which to save the recorder buffers. If *CaptureFileName* is not specified, the recorder buffers are saved in *DriverName.etl*.

DLL

Rcdrkd.dll

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrsearchpath

Article • 10/25/2023

The **!rcdrkd.rcdrsearchpath** extension sets the search path for trace message format (TMF) and trace message control (TMC) files.

dbgcmd

```
!rcdrkd.rcdrsearchpath FilePath
```

Parameters

FilePath

Path to the format files.

DLL

Rcdrkd.dll

Remarks

The search path set by this command takes precedence over the search path specified in the TRACE_FORMAT_SEARCH_PATH environment variable.

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrsettraceprefix

Article • 10/25/2023

The `!rcdrkd.rcdrsettraceprefix` extension sets the trace message prefix.

```
dbgcmd
```

```
!rcdrkd.rcdrsettraceprefix TracePrefixString
```

Parameters

TracePrefixString

The trace message prefix string.

DLL

Rcdrkd.dll

Remarks

Each message in a recorder log has a prefix that you can control by specifying a trace message prefix string. For more information, see [Trace Message Prefix](#).

Examples

In the following example, the trace message prefix is originally `%7!u!: %!FUNC! -`. The parameter `%7!u!` specifies that the prefix includes the message sequence number. The parameter `%!FUNC!` specifies that the prefix includes the name of the function that generated the message. The example calls `!rcdrsettraceprefix` to change the prefix string to `%7!u!`. After that, the log display includes message sequence numbers, but does not include function names.

```
dbgcmd
```

```
0: kd> !rcdrlogdump USBXHCI -a 0xfffffa8010737b60
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci
```

```
.pdb
--- start of log ---
405: TransferRing_DispatchEventsAndReleaseLock - 1.8.1: Mapping Begin : Path
1 TransferRingState_Mapping Events 0x00000000
406: TransferRing_StageRetrieveFromRequest - 1.8.1: WdfRequest
0x0000057FEE117A88 TransferData 0xFFFFFA8011EE8700 Function 0x9 Length 500
TransferSize 500 BytesProcessed 0
...
---- end of log ----

0: kd> !rcdrsettraceprefix %7!u!:
SetTracePrefix: "%7!u!:" 
0: kd> !rcdrlogdump USBXHCI -a 0xfffffa8010737b60
Trace searchpath is:

Trace format prefix is: %7!u!:
Trying to extract TMF information from -
C:\ProgramData\dbg\sym\usbxhci.pdb\D4C85D5D3E2843879EDE226A334D69552\usbxhci
.pdb
--- start of log ---
405: 1.8.1: Mapping Begin : Path 1 TransferRingState_Mapping Events
0x00000000
406: 1.8.1: WdfRequest 0x0000057FEE117A88 TransferData 0xFFFFFA8011EE8700
Function 0x9 Length 500 TransferSize 500 BytesProcessed 0
...
---- end of log ----
```

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrtmffile

Article • 10/25/2023

The **!rcdrkd.rcdrtmffile** extension sets or clears the name of the trace message format (TMF) file.

dbgcmd

```
!rcdrkd.rcdrtmffile [Filename]
```

Parameters

Filename

The name of the TMF file. If this parameter is not specified, the filename is cleared.

DLL

Rcdrkd.dll

See also

[RCDRKD Extensions](#)

!rcdrkd.rcdrtraceprtdebug

Article • 10/25/2023

The **!rcdrkd.rcdrtraceprtdebug** extension turns TracePrt diagnostic mode on or off. This extension should be used under the direction of support.

dbgcmd

```
!rcdrkd.rcdrtraceprtdebug {on|off}
```

Parameters

on

Turns TracePrt diagnostic mode on.

off

Turns TracePrt diagnostic mode off.

DLL

Rcdrkd.dll

See also

[RCDRKD Extensions](#)

HID Extensions

Article • 10/25/2023

This section describes the Human Interface Device (HID) debugger extension commands.

The HID debugger extension commands are implemented in Hidkd.dll. To load the HID commands, enter `.load hidkd.dll` in the debugger.

Getting started with the HID extensions

To start debugging a HID issue, enter the [!hidtree](#) command. The `!hidtree` command displays a list of commands and addresses that you can use to investigate device objects, prepared HID data, and HID report descriptors.

In this section

Topic	Description
!hidkd.help	The <code>!hidkd.help</code> command displays help for the HID debugger extension commands.
!hidkd.hidfdo	The <code>!hidkd.hidfdo</code> command displays HID information associated with a functional device object (FDO).
!hidkd.hidpdo	The <code>!hidkd.hidpdo</code> command displays HID information associated with a physical device object (PDO).
!hidkd.hidtree	The <code>!hidkd.hidtree</code> command displays a list of all device nodes that have a HID function driver along with their child nodes. The child nodes have a physical device object (PDO) that was created by the parent node's HID function driver.
!hidkd.hidppd	The <code>!hidkd.hidppd</code> command displays HID prepared data.
!hidkd.hidrd	The <code>!hidkd.hidrd</code> command displays a HID report descriptor in both raw and parsed format.

See also

[RCDRKD Extensions](#)

[Specialized Extension Commands](#)

!hidkd.help

Article • 04/03/2024

The **!hidkd.help** extension displays help for the HID debugger extension commands.

DLL

Hidkd.dll

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!hidkd.hidfdo

Article • 04/03/2024

The **!hidkd.hidfdo** extension command displays HID information associated with a functional device object (FDO).

```
dbgcmd
```

```
!hidkd.hidfdo fdo
```

Parameters

fdo

Address of an FDO. To get the addresses of FDOs that are associated with HID drivers, use the [!usbhid.hidtree](#) command.

DLL

Hidkd.dll

Examples

Here is an example of the output of the **!hidfdo** command. The example first calls [!hidtree](#) to get the address of an FDO.

```
dbgcmd
```

```
0: kd> !hidkd.hidtree
HID Device Tree
...
FDO  VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
!hidfdo 0xfffffe00004f466e0
...
0: kd> !hidfdo 0xfffffe00004f466e0
# FDO 0xfffffe00004f466e0  (!devobj!/devstack)

      Name          : \Device\_HID00000002
      Vendor ID    : 0x045E(Microsoft Corporation)
      Product ID   : 0x0745
      Version Number: 0x0634
      Is Present?  : Y
      Report Descriptor: !hidrd 0xfffffe00004281a80 0x127
      Per-FDO IFR Log  : !rcdrlogdump HIDCLASS -a 0xFFFFE0000594D000
```

Position in HID tree

dt FDO_EXTENSION 0xfffffe00004f46850

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!hidkd.hidpdo

Article • 04/03/2024

The **!hidkd.hidpdo** extension command displays HID information associated with a physical device object (PDO).

```
dbgcmd
```

```
!hidkd.hidpdo pdo
```

Parameters

pdo

Address of a PDO. To get the addresses of PDOs that are associated with HID drivers, use the [!usbhid.hidtree](#) command.

DLL

Hidkd.dll

Examples

Here is an example of the output of the **!hidpdo** command. The example first calls [!hidtree](#) to get the address of a PDO.

```
dbgcmd
```

```
0: kd> !hidkd.hidtree
HID Device Tree
...
FDO  VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
...
    PDO  Generic Desktop Controls (0x01) | Mouse (0x02)
        !hidpdo 0xfffffe000056281e0
        ...
0: kd> !hidpdo 0xfffffe000056281e0
# PDO 0xfffffe000056281e0  (!devobj!/devstack)

    Collection Num   : 1
    Name             : \Device\_HID00000002#COLLECTION00000001
    FDO              : !hidfdo 0xfffffe00004f466e0
    Per-FDO IFR Log : !rcdrlogdump HIDCLASS -a 0xFFFFE0000594D000
```

```
Usage Page      : Generic Desktop Controls (0x01)
Usage          : Mouse (0x02)
Report Length   : 0xa(Input) 0x0(Output) 0x2(Feature)
Pre-parsed Data : 0xfffffe00003742840
Position in HID tree

dt PDO_EXTENSION 0xfffffe00005628350

## Power States

Power States : S0/D0
Wait Wake IRP : !irp 0xfffffe00004fc57d0 (pending on \Driver\HidUsb)
```

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!hidkd.hidtree

Article • 04/03/2024

The `!hidkd.hidtree` extension command displays a list of all device nodes that have a HID function driver along with their child nodes. The child nodes have a physical device object (PDO) that was created by the parent node's HID function driver.

```
dbgcmd
!hidkd.hidtree
```

This screen shot shows an example of the output of the `!hidtree` command.

```
FDO VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
!hidfdo 0xfffffe0000344d060
PowerStates: S0/D0 | HidSmStateD0 (On2009)
dt FDO_EXTENSION 0xfffffe0000344d1d0
!devnode 0xfffffe00003b18d30 | DeviceNodeStarted (0n776)
InstancePath: USB\VID_045E&PID_0745&MI_01\6&36578c0a&0&0001
IFR Log: !_rcdrlogdump HIDCLASS -a 0xFFFFE00004688000

PDO Generic Desktop Controls (0x01) | Mouse (0x02)
!hidpdo 0xfffffe00004fba6e0
PowerStates:S0/D0 | COLLECTION_STATE_RUNNING (0n3)
dt PDO_EXTENSION 0xfffffe00004fba850
!devnode 0xfffffe00004166720 | DeviceNodeStarted (0n776)
InstancePath:HID\VID_045E&PID_0745&MI_01&Co101\7&1195b533&0&0000

PDO Consumer (0x0C) | Consumer Control (0x01)
!hidpdo 0xfffffe00004f466e0
PowerStates:S0/D0 | COLLECTION_STATE_RUNNING (0n3)
dt PDO_EXTENSION 0xfffffe00004f46850
!devnode 0xfffffe000055bc9f0 | DeviceNodeStarted (0n776)
InstancePath:HID\VID_045E&PID_0745&MI_01&Co102\7&1195b533&0&0001

FDO VendorID:0x045E(Microsoft Corporation) ProductID:0x0745 Version:0x0634
!hidfdo 0xfffffe00001360240
PowerStates: S0/D0 | HidSmStateD0 (On2009)
dt FDO_EXTENSION 0xfffffe000013603b0
!devnode 0xfffffe0000521ba70 | DeviceNodeStarted (0n776)
InstancePath: USB\VID_045E&PID_0745&MI_00\6&36578c0a&0&0000
IFR Log: !_rcdrlogdump HIDCLASS -a 0xFFFFE00004670000

PDO Generic Desktop Controls (0x01) | Keyboard (0x06)
!hidpdo 0xfffffe0000559a6e0
PowerStates:S0/D0 | COLLECTION_STATE_RUNNING (0n3)
dt PDO_EXTENSION 0xfffffe0000559a850
!devnode 0xfffffe0000224e180 | DeviceNodeStarted (0n776)
InstancePath:HID\VID_045E&PID_0745&MI_00\7&29594178&0&0000
```

In this example, there are two device nodes that have a HID function driver. A functional device object (FDO) represents the HID driver in those two nodes. The first FDO node has two child nodes, and the second FDO node has one child node. In the debugger output, the child nodes have the PDO heading.

Note This set of device nodes does not form a tree that has a single root node. The device nodes that have HID function drivers can be isolated from each other.

When you are debugging a HID issue, the `!hidtree` is a good place to start, because the command displays several addresses that you can pass to other HID debugger commands. The output uses [Debugger Markup Language \(DML\)](#) to provide links. The links execute commands that give detailed information related to an individual device node. For example, you could get information about an FDO by clicking one of the `!hidfdo` links. As an alternative to clicking a link, you can enter a command. For example, to see detailed information about the first node in the preceding output, you could enter the command `!devnode 0xffffe00003b18d30`.

Note The DML feature is available in WinDbg, but not in Visual Studio or KD.

DLL

Hidkd.dll

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!hidkd.hidppd

Article • 04/03/2024

The **!hidkd.hidppd** extension command displays HID prepared data.

```
dbgcmd
```

```
!hidkd.hidppd ppd
```

Parameters

ppd

Address of a **HIDP_PREPARSED_DATA** structure. To get the address of a **HIDP_PREPARSED_DATA** structure, use [!hidpdo](#) or [!hidpdo](#).

DLL

Hidkd.dll

Examples

This example shows how to use [!hidpdo](#) followed by [!hidppd](#). The output of [!hidpdo](#) shows the address of a **HIDP_PREPARSED_DATA** structure.

```
dbgcmd
```

```
0: kd> !hidpdo 0xfffffe000029f6060
## PDO 0xfffffe000029f6060  (!devobj!/devstack)

Collection Num   : 1
Name             : \Device\_HID00000000#COLLECTION00000001
...
Pre-parsed Data : 0xfffffe000029d1010
...

0: kd> !hidkd.hidppd 0xfffffe000029d1010
Reading prepared data...
Prepared Data at 0xfffffe000029d1010

## Summary

UsagePage          : Vendor-defined (0xFFA0)
Usage              : 0x01
```

```
Report Length          : 0x2(Input) 0x2(Output) 0x0(Feature)
Link Collection Nodes : 2
Button Caps           : 0(Input) 0(Output) 0(Feature)
Value Caps            : 1(Input) 1(Output) 0(Feature)
Data Indices          : 1(Input) 1(Output) 0(Feature)

## Input Value Capability #0

Report ID             : 0x0
Usage Page            : Vendor-defined (0xFFA1)
Bit Size              : 0x8
Report Count          : 0x1
...
```

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

!hidkd.hidrd

Article • 04/03/2024

The **!hidkd.hidrd** extension command displays a HID report descriptor in both raw and parsed format.

```
dbgcmd
```

```
!hidkd.hidrd rd Length
```

Parameters

rd

Address of the raw report descriptor data. To get the address of the descriptor data, use the [!hidfdo](#) command.

Length

The length, in bytes, of the raw report descriptor data. To get the length, use the [!hidfdo](#) command.

DLL

Hidkd.dll

Examples

This example shows how to use the [!hidfdo](#) command followed by the [!hidrd](#) command. The output of [!hidfdo](#) shows both the address and length of the raw report descriptor data.

```
dbgcmd
```

```
0: kd> !hidfdo 0xfffffe00004f466e0
# FDO 0xfffffe00004f466e0  (!devobj!/devstack)

Name          : \Device\_HID00000002
...
Report Descriptor : !hidrd 0xfffffe00004281a80 0x127
...
0: kd> !hidrd 0xfffffe00004281a80 0x127
Report Descriptor at 0xfffffe00004281a80
```

```
## Raw Data

0x0000: 05 01 09 02 A1 01 05 01-09 02 A1 02 85 1A 09 01
0x0010: A1 00 05 09 19 01 29 05-95 05 75 01 15 00 25 01
0x0020: 81 02 75 03 95 01 81 01-05 01 09 30 09 31 95 02
...
## Parsed

Usage Page (Generic Desktop Controls).....0x0000: 05 01
Usage (Mouse).....0x0002: 09 02
Collection (Application).....0x0004: A1 01
..Usage Page (Generic Desktop Controls).....0x0006: 05 01
..Usage (Mouse).....0x0008: 09 02
..Collection (Logical).....0x000A: A1 02
....Report ID (26).....0x000C: 85 1A
...
End Collection ().....0x0126: C0
```

See also

[HID Extensions](#)

Feedback

Was this page helpful?

 Yes

 No

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

Logger Extensions (Logexts.dll)

Article • 10/25/2023

Extension commands related to the Logger and LogViewer tools can be found in Logexts.dll.

This DLL appears in the winext directory. It can be used with all Windows operating systems.

!logexts.help

Article • 04/03/2024

The **!logexts.help** extension displays a Help text in the Command Prompt window showing all Logexts.dll extension commands.

```
dbgcmd
!logexts.help
```

DLL

Logexts.dll

Additional Information

For more information, see [Logger](#) and [LogViewer](#).

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.logb

Article • 04/03/2024

The **!logexts.logb** extension displays or flushes the output buffer.

dbgcmd

```
!logexts.logb p  
!logexts.logb f
```

Parameters

p

Causes the contents of the output buffer to be displayed in the debugger.

f

Flushes the output buffer to the disk.

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

As a performance consideration, log output is flushed to disk only when the output buffer is full. By default, the buffer is 2144 bytes.

The **!logexts.logb p** extension displays the contents of the buffer in the debugger.

The **!logexts.logb f** extension flushes the buffer to the log files. Because the buffer memory is managed by the target application, the automatic writing of the buffer to disk will not occur if there is an access violation or some other nonrecoverable error in the target application. In such cases, you should use this command to manually flush the buffer to the disk. Otherwise, the most recently-logged APIs might not appear in the log files.

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.logc

Article • 04/03/2024

The **!logexts.logc** extension displays all API categories, displays all APIs in a specific category, or enables and disables the logging of APIs in one or more categories.

dbgcmd

```
!logexts.logc e Categories
!logexts.logc d Categories
!logexts.logc p Category
!logexts.logc
```

Parameters

e

Enables logging of the specified categories.

d

Disables logging of the specified categories.

Categories

Specifies the categories to be enabled or disabled. If multiple categories are listed, separate them with spaces. An asterisk (*) can be used to indicate all categories.

p

Displays all APIs that belong to the specified category.

Category

Specifies the category whose APIs will be displayed. Only one category can be specified with the **p** option.

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

Without any options, `!logexts.logc` will display the current list of available categories and will indicate which ones are enabled and disabled.

If a category is disabled, the hooks for all APIs in that category will be removed so there is no longer any performance overhead. COM hooks are not removed, because they cannot be re-enabled at will.

Enabling only certain categories can be useful when you are only interested in a particular type of interaction that the program is having with Windows (for example, file operations). This reduces the log file size and also reduces the effect that Logger has on the execution speed of the process.

The following command will enable the logging of all categories:

```
dbgcmd  
0:000> !logexts.logc e *
```

The following command will disable the logging of category 7:

```
dbgcmd  
0:000> !logexts.logc d 7
```

The following command will enable the logging of categories 13 and 15:

```
dbgcmd  
0:000> !logexts.logc e 13 15
```

The following command will display all APIs belonging to category 3:

```
dbgcmd  
0:000> !logexts.logc p 3
```

Feedback

Was this page helpful?

 Yes

 No

!logexts.logd

Article • 04/03/2024

The `!logexts.logd` extension disables logging.

```
dbgcmd
!logexts.logd
```

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

This will cause all API hooks to be removed in an effort to allow the program to run freely. COM hooks are not removed, because they cannot be re-enabled at will.

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.loge

Article • 04/03/2024

The **!logexts.loge** extension enables logging. If logging has not been initialized, it will be initialized and enabled.

dbgcmd

```
!logexts.loge [OutputDirectory]
```

Parameters

OutputDirectory

Specifies the directory to use for output. If *OutputDirectory* is specified, it must exist -- the debugger will not create it. If a relative path is specified, it will be relative to the directory from which the debugger was started. If *OutputDirectory* is omitted, the path to the Desktop will be used. The debugger will create a LogExts subdirectory of *OutputDirectory*, and all Logger output will be placed in this subdirectory.

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

If Logger has not yet been injected into the target application by the [!logexts.logi](#) extension, the **!logexts.loge** extension will inject Logger into the target and then enable logging.

If [!logexts.logi](#) has already been run, you cannot include *OutputDirectory*, because the output directory will have already been set.

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.logi

Article • 04/03/2024

The **!logexts.logi** extension initializes logging by injecting Logger into the target application.

dbgcmd

```
!logexts.logi [OutputDirectory]
```

Parameters

OutputDirectory

Specifies the directory to use for output. If *OutputDirectory* is specified, it must exist -- the debugger will not create it. If a relative path is specified, it will be relative to the directory from which the debugger was started. If *OutputDirectory* is omitted, the path to the Desktop will be used. The debugger will create a LogExts subdirectory of *OutputDirectory*, and all Logger output will be placed in this subdirectory.

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

This command initializes logging, but does not actually enable it. Logging can be enabled with the [!logexts.loe](#) command.

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.logm

Article • 04/03/2024

The **!logexts.logm** extension creates or displays a module inclusion list or a module exclusion list.

dbgcmd

```
!logexts.logm i Modules
!logexts.logm x Modules
!logexts.logm
```

Parameters

i

Causes Logger to use a module inclusion list. It will consist of the specified *Modules*.

x

Causes Logger to use a module exclusion list. It will consist of Logexts.dll, kernel32.dll, and the specified *Modules*.

Modules

Specifies the modules to be included or excluded. This list is not cumulative; each use of this command creates an entirely new list. If multiple modules are listed, separate them with spaces. An asterisk (*) can be used to indicate all modules.

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

With no parameters, the **!logexts.logm** extension displays the current inclusion list or exclusion list.

The extensions **!logexts.logm x *** and **!logexts.logm i** are equivalent: they result in a completely empty inclusion list.

The extensions **!logexts.logm i *** and **!logexts.logm x** are equivalent: they result in an exclusion list that contains only Logexts.dll and kernel32.dll. These two modules are always excluded, because Logger is not permitted to log itself.

Here are some examples:

```
dbgcmd

0:001> !logm
Excluded modules:
    LOGEXTS.DLL      [mandatory]
    KERNEL32.DLL      [mandatory]
    USER32.DLL
    GDI32.DLL
    ADVAPI32.DLL

0:001> !logm x winmine.exe
Excluded modules:
    Logexts.dll      [mandatory]
    kernel32.dll      [mandatory]
    winmine.exe

0:001> !logm x user32.dll gdi32.dll
Excluded modules:
    Logexts.dll      [mandatory]
    kernel32.dll      [mandatory]
    user32.dll
    gdi32.dll

0:001> !logm i winmine.exe mymodule2.dll
Included modules:
    winmine.exe
    mymodule2.dll
```

Feedback

Was this page helpful?

 Yes

 No

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

!logexts.logo

Article • 04/03/2024

The **!logexts.logo** extension sets or displays the Logger output options.

dbgcmd

```
!logexts.logo {e|d} {d|t|v}
!logexts.logo
```

Parameters

e|d

Specifies whether to enable (e) or disable (d) the indicated output type.

d|t|v

Specifies the output type. Three types of Logger output are possible: messages sent directly to the debugger (d), a text file (t), or a verbose .lgv file (v).

DLL

Logexts.dll

Additional Information

For more information, see [Logger and LogViewer](#).

Remarks

If **!logexts.logo** is used without any parameters, then the current logging status, the output directory, and the current settings for the debugger, text file, and verbose log are displayed:

dbgcmd

```
0:000> !logo
Logging currently enabled.

Output directory: MyLogs\LogExts\
Output settings:
```

Debugger	Disabled
Text file	Enabled
Verbose log	Enabled

In the previous example, the output directory is a relative path, so it is located relative to the directory in which the debuggers were started.

To disable verbose logging, you would use the following command:

dbgcmd	
0:000> !logo d v	
Debugger	Disabled
Text file	Enabled
Verbose log	Disabled

Text file and .lgv files will be placed in the current output directory. To read an .lgv file, use LogViewer.

Feedback

Was this page helpful?

 Yes

 No

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

NDIS Extensions (Ndiskd.dll)

Article • 10/25/2023

This section describes commands available in !ndiskd, a debugger extension that is useful for debugging NDIS (Network Device Interface Specification) drivers. These commands enable network driver developers to see a bigger picture of the Windows networking stack and how their drivers interact with it. With !ndiskd, you can see the state of all network adapters ([!ndiskd.netadapter](#)), a visual diagram of the computer's network stack ([!ndiskd.netreport](#)), a log of traffic on the network adapters([!ndiskd.nbllog](#)), or a list of all pending OID requests ([!ndiskd.oid](#)).

The commands can be found in Ndiskd.dll. To load the symbols, enter `.reload /f ndis.sys` in the debugger command window. To confirm the symbols loaded successfully, use the [!lmi ndis](#) extension and look for the phrase "Symbols loaded successfully" toward the bottom. Your output should look similar to the following example:

```
dbgcmd

0: kd> !lmi ndis
Loaded Module Info: [ndis]
    Module: ndis
    Base Address: fffff80174570000
    Image Name: ndis.sys
    Machine Type: 34404 (X64)
    Time Stamp: 938f9f4e (This is a reproducible build file hash, not a
true timestamp)
    Size: 16f000
    CheckSum: 167a05
    Characteristics: 22
    Debug Data Dirs: Type      Size      VA      Pointer
                      CODEVIEW   21, d4060,   d2c60 RSDS - GUID: {9CC82DBE-96A0-
773D-29E0-62B698C4C3A8}
                      Age: 1, Pdb: ndis.pdb
                      POGO     988, d4084,   d2c84 [Data not mapped]
                      REPRO    24, d4a0c,   d360c Reproducible build[Data not
mapped]
                      Image Type: MEMORY      - Image read successfully from loaded memory.
                      Symbol Type: PDB        - Symbols loaded successfully from symbol server.

C:\ProgramData\Txt\sym\ndis.pdb\9CC82DBE96A0773D29E062B698C4C3A81\ndis.pdb
Load Report: public symbols , not source indexed

C:\ProgramData\Txt\sym\ndis.pdb\9CC82DBE96A0773D29E062B698C4C3A81\ndis.pdb
```

!ndiskd Hyperlinks

Many of the extension commands in !ndiskd present you with hyperlinks in the results they display in the debugger window. The text for these hyperlinks has been left in the samples provided to illustrate the exact format of what you will see when you run the command on your debuggee machine. Some of the examples also refer explicitly to clicking on these links so you can understand typical usage flows, though the examples also provide the alternate command line forms of each command.

Common Parameters

All !ndiskd commands support the following generic parameters.

-verbose

Shows additional details.

-terse

Suppresses some boilerplate output.

-static

Suppresses some interactive output.

-dml 0|1

Controls whether DML (debugger markup language) output is enabled.

-unicode 0|1

Controls whether Unicode character output is allowed.

-indent N

Uses *N* spaces per level of indent.

-force

Overrides some safety checks on remote data sanity.

-tracedata

Shows verbose trace messages to debug !ndiskd itself.

Net Adapter, NDIS Driver, and General Commands

The following commands display information about the machine's network adapters, network drivers, and general commands associated with the network stack (such as rcvqueues, opens, filters, OIDs, and RW locks).

- [!ndiskd.netadapter](#)

- [!ndiskd.minidriver](#)
- [!ndiskd.rcvqueue](#)
- [!ndiskd.protocol](#)
- [!ndiskd.mopen](#)
- [!ndiskd.filter](#)
- [!ndiskd.filterdriver](#)
- [!ndiskd.oid](#)
- [!ndiskd.ndisrwlock](#)
- [!ndiskd.netreport](#)

NET_BUFFER_LIST and NET_BUFFER Commands

The following commands display information relating to [NET_BUFFER_LIST](#) and [NET_BUFFER](#) structures.

- [!ndiskd.nbl](#)
- [!ndiskd.nb](#)
- [!ndiskd.nblpool](#)
- [!ndiskd.nbpool](#)
- [!ndiskd.pendingnbls](#)
- [!ndiskd.nbllog](#)

NetAdapterCx Commands

The following commands display information relating to the Network Adapter WDF Class Extension [NetAdapterCx](#) and its associated structures, [NET_RING_BUFFER](#) and [NET_PACKET](#).

- [!ndiskd.cxadapter](#)
- [!ndiskd.netqueue](#)
- [!ndiskd.netrb](#)
- [!ndiskd.netpacket](#)
- [!ndiskd.netfragment](#)
- [!ndiskd.nrc](#)
- [!ndiskd.netring](#)

Network Interface Commands

The following commands display information relating to network interfaces.

- [!ndiskd.interfaces](#)

- [!ndiskd.ifprovider](#)

NDIS_PACKET Commands

The following commands display information about [NDIS_PACKET](#) structures. These extensions are for legacy NDIS 5.x drivers. The NDIS_PACKET structure and its associated architecture have been deprecated.

- [!ndiskd.pkt](#)
- [!ndiskd.pktpools](#)
- [!ndiskd.findpacket](#)

CoNDIS Commands

The following commands display information about [Connection-Oriented NDIS](#) connections.

- [!ndiskd.vc](#)
- [!ndiskd.af](#)

NDIS Debugging Commands

The following commands display information relating to NDIS refcounts, event logs, stack traces, and debug traces.

- [!ndiskd.ndisref](#)
- [!ndiskd.ndisevent](#)
- [!ndiskd.ndisstack](#)
- [!ndiskd.dbglevel](#)
- [!ndiskd.dbgsystems](#)

WDI Commands

The following commands display information about [WDI Miniport Drivers](#).

- [!ndiskd.wdiadapter](#)
- [!ndiskd.wdiminidriver](#)
- [!ndiskd.nwadapter](#)

NDIS and !ndiskd Information Commands

The following commands display information about NDIS.sys and ndiskd.dll.

- [!ndiskd.ndis](#)
- [!ndiskd.ndiskdversion](#)

Miscellaneous Commands

- [!ndiskd.ifstacktable](#)
- [!ndiskd.compartments](#)
- [!ndiskd.ndisslot](#)

Related Topics

For more information about designing NDIS drivers for Windows Vista and later, see the [Network Driver Design Guide](#).

For more information about reference for NDIS drivers for Windows Vista and later, see [Windows Vista and Later Networking Reference](#).

For a demonstration of using the !ndiskd debugger commands to debug the network stack, see the [Debugging the Network Stack](#) channel 9 video.

!ndiskd.help

Article • 10/25/2023

The **!ndiskd.help** command displays a list of available !ndiskd commands with a brief description of each one.

```
Console
!ndiskd.help
```

DLL

Ndiskd.dll

Examples

The following example shows the list of help commands using **!ndiskd.help**.

```
Console
3: kd> !ndiskd.help

NDIS KD EXTENSIONS

    help          This help and lots more
    netadapter    Show network adapters (this is a good starting
place)
    protocol     Show protocol drivers
    mopen         Show open bindings between netadapter and protocol
    filter        Show light-weight filters
    nbl           Show information about an NET_BUFFER_LIST
    oid           Show pending OID Requests
    ndis          Show NDIS.sys build info
    netreport    Draw a box diagram of your network stack

    Show more extensions
    View examples & tutorials online
```

By using **!ndiskd.help -all**, you'll get a more detailed list, as shown in the following example.

Note

Some alternate commands are listed at the bottom of this example. These commands

are available for NDIS driver developers who have used them before but we recommend using the primary commands instead.

Console

```
3: kd> !ndiskd.help -all
```

NDIS KD EXTENSIONS

Primary commands:

help	This help and lots more
netadapter	Show network adapters (this is a good starting place)
minidriver	Show network adapter drivers
rcvqueue	Show a receive queue (c.f. VMQ)
protocol	Show protocol drivers
mopen	Show open bindings between netadapter and protocol
filter	Show light-weight filters
filterdriver	Show filter drivers (not to be confused with filters)
nbl	Show information about an NET_BUFFER_LIST
nb	Show information about an NET_BUFFER
nblpool	Show information about an NET_BUFFER_LIST pool
nbpool	Show information about an NET_BUFFER pool
pendingnbls	Show all NET_BUFFER_LISTs that are in transit
nbllog	Show a log of all NET_BUFFER_LIST activity
oid	Show pending OID Requests
interfaces	Show interfaces (à la NDISIF)
ifprovider	Show registered NDIS interface providers
ifstacktable	Show the ifStackTable
networks	Show networks (probably not what you think)
compartments	Show compartments
pkt	Show a NDIS_PACKET structure
pktpools	Show all allocated packet pools
findpacket	Find a packet in memory
vc	Show a CoNDIS virtual connection
af	Show a CoNDIS address family
ndisref	Show a debug log of refcounts
ndisevent	Show a debug log of events
ndisstack	Show a debug stack trace
wdiadapter	Shows one or more WDIWiFi!CAdapter structures
wdiminidriver	Shows one or more CMiniportDriver structures
nwadapter	Shows one or more nwifi!ADAPT structures
ndisrwlock	Show an NDIS_RW_LOCK_EX lock
dbglevel	Change the debugging level [checked NDIS.sys only]
dbgsystems	Toggle subsystems being debugged [checked NDIS.sys only]
ndiskdversion	Show info about NDISKD itself
netreport	Draw a box diagram of your network stack
cxadapter	Show information about an NETADAPTER
netqueue	Show information about a NetAdapterCx datapath queue
nrc	Show information about an NET_RING_COLLECTION
netring	Show information about an NET_RING
netpacket	Show information about an NET_PACKET
netfragment	Show information about an NET_FRAGMENT

Alternate commands:

miniport	Same as !ndiskd.netadapter
gminiports	Same as !ndiskd.netadapter
miniports	"Classic" version of !ndiskd.netadapter
filterdb	Same as !ndiskd.netadapter -filterdb
offload	Same as !ndiskd.netadapter -offloads
ports	Same as !ndiskd.netadapter -ports
rcvqueues	Same as !ndiskd.netadapter -rcvqueues
filters	Same as !ndiskd.filter
opens	Same as !ndiskd.mopen
protocols	Same as !ndiskd.protocol
nblpools	Same as !ndiskd.nblpool
nbpools	Same as !ndiskd.nbpool

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

!ndiskd.netadapter

Article • 10/25/2023

The **!ndiskd.netadapter** extension displays information about NDIS miniports, or network adapters, that are active on the system. If you run this command with no parameters, !ndiskd will display a list of all network adapters.

Console

```
!ndiskd.netadapter [-handle <x>] [-basic] [-diag] [-state] [-bindings]
    [-ports] [-offloads] [-filterdb] [-timers] [-rst]
    [-pm] [-ss] [-aoac] [-wol] [-protocoloffloads]
    [-rss] [-hw] [-device] [-wmi] [-customwmi]
    [-ndiswmi] [-ref] [-log] [-grovel] [-findname <any>]
    [-rcvfilter] [-nicswitch] [-rcvqueues] [-nicswitches] [-iov]
    [-vfs] [-vports] [-iftrace] [-ip]
```

Parameters

-handle

Handle of an NDIS miniport.

-basic

Displays summary information about the miniport.

-diag

Displays auto-diagnostic alerts (if any).

-state

Displays the miniport's current state.

-bindings

Displays miniport bindings.

-ports

Shows a list of NDIS ports.

-offloads

Shows task offload state and capabilities.

-filterdb

Shows the current packet filter.

-timers

Shows timer objects allocated by the miniport.

-rst

Shows Receive-Side Throttling state.

-pm

Shows power management state and capabilities.

-ss

Shows Selective Suspend state.

-aoac

Shows AOAC (Connected Standby) state.

-wol

Shows Wake-on-LAN (WoL) configuration.

-protocoloffloads

Shows active power management protocol offloads.

-rss

Shows Receive Side Scaling parameters.

-hw

Displays hardware resources.

-device

Shows information about the underlying NT device object.

-wmi

Shows WMI GUIDs registered to the adapter.

-customwmi

Shows custom WMI GUIDs registered by the miniport.

-ndiswmi

Shows NDIS-provided WMI GUIDs.

-ref

Shows a breakdown of references on the miniport.

-log

Displays a PnP and Power event log.

-grovel

Forces a search for miniport blocks in memory.

-findname

Filters miniports by name prefix.

-rcvfilter

Shows receive filtering capabilities.

-nicswitch

Shows NIC switch capabilities.

-rcvqueues

Shows receive queues.

-nicswitches

Shows NIC switches.

-iov

Shows SR-IOV (Single Root I/O Virtualization) capabilities.

-vfs

Shows SR-IOV VFs (Virtual Filters).

-vports

Shows Vports (Virtual ports).

-ifrtrace

Shows the in-flight recorder's trace.

-ip

Shows IP addresses on the network's interface.

DLL

Ndiskd.dll

Examples

By running **!ndiskd.netadapter** with no parameters, you can get a list of all network adapters on the system along with their associated miniport drivers. In this example output, look for the Microsoft Kernel Debug Network Adapter, whose handle is fffffdf80140c71a0. For more information about what the Kernel Debug Network Adapter is, see [Kernel debugging over the network](#) on the NDIS blog.

Console

```
3: kd> !ndiskd.netadapter
   Driver          NetAdapter      Name
   fffffdf8015a98380  fffffdf8015aa11a0  Microsoft ISATAP Adapter #2
   fffffdf801418d650  fffffdf80140c71a0  Microsoft Kernel Debug Network
   Adapter
```

By clicking on the handle for the miniport driver or entering the `!ndiskd.netadapter - handle`, you can now see all of NDIS's state on that device. This can be helpful very as a starting place for troubleshooting a network driver or for figuring out where an issue is in the network stack. For example, you can see the Datapath state for the driver and see whether it is connected or not.

At the bottom of the report for this net adapter, there are many other links you can click on to explore further information, such as any pending OIDs and the state of task offloads. These links correspond to many of the parameters for `!ndiskd.netadapter`.

Console

```
3: kd> !ndiskd.netadapter fffffdf80140c71a0
```

MINIPORT

Microsoft Kernel Debug Network Adapter

Ndis handle	fffffdf80140c71a0
Ndis API version	v6.20
Adapter context	fffffdf80147d7230
Driver	fffffdf801418d650 - kdnic v4.2
Network interface	fffffdf80139b3a20
Media type	802.3
Physical medium	NdisPhysicalMediumOther
Device instance	ROOT\KDNIC\0000
Device object	fffffdf80140c7050 More information
MAC address	18-03-73-c1-e8-72

STATE

Miniport	Running
Device PnP	Started Show state history
Datapath	Normal
Interface	Up
Media	Connected
Power	D0
References	0n10 Show detail
Total resets	0

Pending OID	None
Flags	NOT_BUS_MASTER, ALLOW_BUGCHECK_CALLBACK, BUGCHECK_CALLBACK_REGISTERED, DEFAULT_PORT_ACTIVATED, SUPPORTS_MEDIA_SENSE, DOES_NOT_DO_LOOPBACK, MEDIA_CONNECTED
PnP flags	VIRTUAL_DEVICE, HIDDEN, NO_HALT_ON_SUSPEND, RECEIVED_START

BINDINGS

Protocol list	Driver	Open	Context
MSLLDP	fffffdf80120a5c10	fffffdf8015a749c0	
fffffdf8015d325e0			
TCPIP	fffffdf80131cc010	fffffdf801494a650	
fffffdf801494aa50			
NDISUIO	fffffdf8015a58140	fffffdf8015a78c10	
fffffdf8015a77e00			
TCPIP6	fffffdf80131c9c10	fffffdf80147875a0	
fffffdf801494f010			
(RASPPPOE)	Not running		
RSPNDR	fffffdf80120a0c10	fffffdf8015a79c10	
fffffdf8015a79010			
LLTDIO	fffffdf8015a5f9b0	fffffdf801406f010	
fffffdf8015a786c0			
(RDMANDK)	fffffdf801406d8f0	Declined with	
NDIS_STATUS_NOT_RECOGNIZED			

Filter list	Driver	Module	Context
WFP 802.3 MAC Layer LightWeight Filter-0000			
	fffffdf80139a5a70	fffffdf801494c670	
fffffdf801494a010			
QoS Packet Scheduler-0000			
	fffffdf8014039d90	fffffdf801494dc70	
fffffdf80147dc2b0			
WFP Native MAC Layer LightWeight Filter-0000			
	fffffdf80139fcd70	fffffdf8014950c70	
fffffdf8014950880			

MORE INFORMATION

Driver handlers	Task offloads
Power management	PM protocol offloads
Pending OIDs	Timers
Pending NBLs	Receive side throttling
Wake-on-LAN (WoL)	Packet filter
Receive queues	Receive filtering
RSS	NIC switch
Hardware resources	Selective suspend
NDIS ports	WMI guides
Diagnostic log	

As example of using **!ndiskd.netadapter** as a starting place for further debugging, click on the "Driver handlers" link at the bottom of the report to see a list of all registered driver callback handlers for this net adapter's miniport driver. In the following example, clicking the link causes !ndiskd to run the **!ndiskd.minidriver** extension with the handle of this net adapter's miniport driver. The miniport driver is the kdnic 4.2 and its handle is fffffdf801418d650.

```
Console

3: kd> !ndiskd.minidriver fffffdf801418d650 -handlers

HANDLERS

    NDIS Handler          Function pointer   Symbol (if
available)
    InitializeHandlerEx  ffffff80f1fd78230  bp
    SetOptionsHandler     ffffff80f1fd72800  bp
    HaltHandlerEx        ffffff80f1fd78040  bp
    ShutdownHandlerEx    ffffff80f1fd722c0  bp

    CheckForHangHandlerEx ffffff80f1fd72810  bp
    ResetHandlerEx       ffffff80f1fd72f70  bp

    PauseHandler         ffffff80f1fd78000  bp
    RestartHandler       ffffff80f1fd78940  bp

    OidRequestHandler   ffffff80f1fd71c90  bp
    CancelOidRequestHandler ffffff80f1fd722c0  bp
    DirectOidRequestHandler [None]
    CancelDirectOidRequestHandler [None]
    DevicePnPEventNotifyHandler ffffff80f1fd789a0  bp

    SendNetBufferListsHandler ffffff80f1fd71870  bp
    ReturnNetBufferListsHandler ffffff80f1fd71b50  bp
    CancelSendHandler     ffffff80f1fd722c0  bp
```

You can now click the "bp" link to the right of each handler to set a breakpoint on that handler to debug a particular problem. For example, if there is a hang in the datapath, you can investigate the driver's SendNetBufferListsHandler or ReturnNetBufferListsHandler.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Kernel debugging over the network

!ndiskd.minidriver

!ndiskd.minidriver

Article • 10/25/2023

The **!ndiskd.minidriver** command displays information about an NDIS miniport driver. If you run this extension with no parameters, !ndiskd will display a list of NDIS miniport drivers that are active on the system.

Console

```
!ndiskd.minidriver [-handle <x>] [-basic] [-miniports] [-devices] [-handlers]
```

Parameters

-handle

Optional handle of an NDIS miniport driver.

-basic

Displays basic information about the miniport driver.

-miniports

Displays the miniports associated with this miniport driver.

-devices

Displays devices associated with this miniport driver.

-handlers

Displays this driver's miniport handlers.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.minidriver** command with no parameters to get a list of all NDIS miniport drivers active on the system. In the following example, look for the knic adapter's handle, fffffd20d12dec020

Console

```
1: kd> !ndiskd.minidriver -basic
fffffd20d173deae0 - tunnel
fffffd20d12dec020 - kdnic
```

Using the handle for the kdnic adapter, you can now click on the handle or enter the **!ndiskd.minidriver -handle** command to see detailed information for the tunnel miniport driver, as well as a list of miniports associated with it.

```
Console

1: kd> !ndiskd.minidriver fffffd20d12dec020

MINIPORT DRIVER

    kdnic

        Ndis handle      fffffd20d12dec020      [type it]
        Driver context   ffffff80d2fa15100
        DRIVER_OBJECT    fffffd20d12dee540
        Driver image     kdnic.sys
        Registry path    \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\kdnic
        Reference Count  2
        Flags            [No flags set]

MINIPORTS

    Miniport
    fffffd20d12dd71a0 - Microsoft Kernel Debug Network Adapter

    Handlers
    Device objects
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.rcvqueue

Article • 10/25/2023

The **!ndiskd.rcvqueue** command displays information about a receive queue.

Console

```
!ndiskd.rcvqueue -handle <x> [-filters] [-mem] [-verbose] [-rcvqueueverbosity <x>]
```

Parameters

-handle

Required. Handle of a receive queue.

-filters

Shows filters on the queue.

-mem

Shows shared memory allocations.

-verbose

Shows additional details.

-rcvqueueverbosity

Level of detail to display.

DLL

Ndiskd.dll

Examples

To obtain the receive queue handle, first enter the [!ndiskd.netadapter](#) command with no parameters to see the list of net adapters, their drivers, and their handles. In the following example, look for the Microsoft ISATAP Adapter #2's NetAdapter handle, ffff8083e02ce1a0.

Console

```
3: kd> !ndiskd.netadapter
   Driver          NetAdapter        Name
   fffff8083e2668970  fffff8083e02ce1a0  Microsoft ISATAP Adapter #2
   fffff8083e210fae0  fffff8083e0f501a0  Microsoft Kernel Debug Network
   Adapter
```

Next, with the net adapter's handle, use the **!ndiskd.netadapter -handle -rcvqueues** command to obtain a list of receive queues for this net adapter along with their handles. In this example, there is only one receive queue (the default one) with a handle of fffff8083e3a3d3a0.

Console

```
3: kd> !ndiskd.netadapter fffff8083e02ce1a0 -rcvqueues
```

RECEIVE QUEUES

QueueId	Queue Handle	Processor Affinity
0 [Default]	fffff8083e3a3d3a0	0:0000000000000000 (group:mask)
	Queue Name:	[Zero-length string]
	VM Name:	[Zero-length string]

Now you can use the queue handle to examine the receive queue details with the **!ndiskd.rcvqueue** command.

Console

```
3: kd> !ndiskd.rcvqueue fffff8083e3a3d3a0
```

RECEIVE QUEUE

[Zero-length string]	
VM name	[Zero-length string]
QueueId	0
Ndis handle	fffff8083e3a3d3a0
Miniport	fffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
Open	[No associated Open]
Type	Unspecified
Flags	[No flags set]
Allocated	Yes
References	1
Num filters	0
Num buffers hint	0

```
MSI-X entry      0
Lookahead size   0
Processor affinity 0:0000000000000000 (group:mask)
```

```
Receive filter list
Shared memory allocations
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[!ndiskd.netadapter](#)

!ndiskd.protocol

Article • 10/25/2023

The **!ndiskd.protocol** command displays information about an NDIS protocol driver. If you run this extension with no parameters, !ndiskd will display a list of NDIS protocol drivers that are active on the system.

Console

```
!ndiskd.protocol [-handle <x>] [-findname <any>]
```

Parameters

-handle

Optional handle of an NDIS protocol.

-findname

Filters protocols by name prefix.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.protocol** command to see a list of all NDIS protocols, their handles, and open bindings to miniports (if any). In the following example, look for the TCPIP6TUNNEL protocol's handle, ffff8083e1a95c00.

Console

```
3: kd> !ndiskd.protocol
ffff8083e0114730 - NDISUIO
ffff8083e55f3010 - Microsoft Kernel Debug Network Adapter

ffff8083e3e90c10 - MSLLDP
ffff8083e3926010 - Microsoft Kernel Debug Network Adapter

ffff8083e3e98c10 - WANARPV6

ffff8083e3e97010 - WANARP

ffff8083e3e8f6b0 - RSPNDR
```

```
fffff8083e11902c0 - Microsoft Kernel Debug Network Adapter  
fffff8083e3e90800 - LLTDIO  
fffff8083e15537d0 - Microsoft Kernel Debug Network Adapter  
fffff8083e1a9ac10 - RDMANDK  
fffff8083e1a95c00 - TCPIP6TUNNEL  
fffff8083e56b8110 - Microsoft ISATAP Adapter #2  
fffff8083e19bec10 - TCPIPTUNNEL  
fffff8083e19bfc10 - TCPIP6  
fffff8083e504c770 - Microsoft Kernel Debug Network Adapter  
fffff8083e11cec10 - TCPIP  
fffff8083e0c565a0 - Microsoft Kernel Debug Network Adapter
```

With the protocol's handle, now you can enter either click the handle or enter the **!ndiskd.protocol -handle** command to see information for that protocol, such as the handles for the miniports that are bound to it.

Console

```
3: kd> !ndiskd.protocol fffff8083e1a95c00
```

PROTOCOL

TCPIP6TUNNEL

```
Ndis handle      fffff8083e1a95c00  
Ndis API version v6.40  
Driver context    ffffff80e2e4f9de0  
Driver version    v0.0  
Reference count   2  
Flags             [No flags set]  
Driver image      [Not available]    Why not?
```

BINDINGS

Open	Miniport	Miniport Name
fffff8083e56b8110	fffff8083e02ce1a0	Microsoft ISATAP Adapter #2

HANDLERS

Protocol handler available	Function pointer	Symbol (if available)
BindAdapterHandlerEx	fffff80e2e3baab0	bp
UnbindAdapterHandlerEx	fffff80e2e3c1c80	bp

OpenAdapterCompleteHandlerEx	fffff80e2e4bc940	bp
CloseAdapterCompleteHandlerEx	fffff80e2e3d19b0	bp
NetPnPEventHandler	fffff80e2e3bb140	bp
UninstallHandler	[None]	
SendNetBufferListsCompleteHandler	fffff80e2e3919a0	bp
ReceiveNetBufferListsHandler	fffff80e2e3918a0	bp
StatusHandlerEx	fffff80e2e3a9550	bp
OidRequestCompleteHandler	fffff80e2e398120	bp
DirectOidRequestCompleteHandler	fffff80e2e398120	bp

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.mopen

Article • 10/25/2023

The **!ndiskd.mopen** extension displays information about bindings between miniports and protocols. If you run this extension with no parameters, !ndiskd will display a list of all open bindings between NDIS miniport drivers and protocol drivers.

Console

```
!ndiskd.mopen [-handle <x>] [-ref]
```

Parameters

-handle

Optional handle of an NDIS open binding.

-ref

Shows refcounts of open bindings.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.mopen** command to get a list of all open bindings. In this example, look for the binding between the Microsoft ISATAP Adapter #2 miniport and the TCPIP6TUNNEL protocol. Its handle is ffff8083e56b8110.

Console

```
3: kd> !ndiskd.mopen
Open ffff8083e56b8110
    Miniport: ffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
    Protocol: ffff8083e1a95c00 - TCPIP6TUNNEL

Open ffff8083e11902c0
    Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
    Protocol: ffff8083e3e8f6b0 - RSPNDR

Open ffff8083e55f3010
    Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
    Protocol: ffff8083e0114730 - NDISUIO
```

```
Open ffff8083e15537d0
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e3e90800 - LLTDIO

Open ffff8083e3926010
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e3e90c10 - MSLLDP

Open ffff8083e0c565a0
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e11cec10 - TCPIP

Open ffff8083e504c770
  Miniport: ffff8083e0f501a0 - Microsoft Kernel Debug Network Adapter
  Protocol: ffff8083e19bfc10 - TCPIP6
```

Now you can either click on the handle or use the handle to enter the **!ndiskd.mopen - handle** command, which enables you to see more details about that open binding such as its Datapath state and Receive path information.

```
Console

3: kd> !ndiskd.mopen ffff8083e56b8110

OPEN

  Ndis handle      ffff8083e56b8110
  Flags            [No flags set]
  References       1                  Show detail
  Source           1
  Datapath state   Running

  Protocol         ffff8083e1a95c00 - TCPIP6TUNNEL
  Protocol context  ffff8083e15b62e0

  Miniport         ffff8083e02ce1a0 - Microsoft ISATAP Adapter #2
  Miniport context  ffff8083e0772010

RECEIVE PATH

  Packet filter    [No flags set]
  Frame Type(s)    0x86dd
```

See also

[Network Driver Design Guide](#)

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

!ndiskd.filter

Article • 10/25/2023

The **!ndiskd.filter** extension displays information about an NDIS light-weight filter (LWF). If you run this extension with no parameters, !ndiskd will display a list of all LWFs.

Console

```
!ndiskd.filter [-handle <x>] [-findname <any>] [-handlers]
```

Parameters

-handle

Optional handle of an NDIS light-weight filter.

-findname

Filters LWFs by name prefix.

-handlers

Displays this LWF's filter handlers.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.filter** command with no parameters to get a list of all filters. In this example, look for the ffff8083e14e8460 handle. Note that this handle is for the filter itself and is nested under its associated filter *driver*, the QoS Packet Scheduler.

Console

```
3: kd> !ndiskd.filter
ffff8083e1a7fd90 - QoS Packet Scheduler
    Filter ffff8083e14e8460, Miniport ffff8083e0f501a0 - Microsoft Kernel
    Debug Network Adapter
    ffff8083e1a96b80 - Virtual WiFi Filter Driver
    ffff8083e19c4b70 - WFP vSwitch Layers LightWeight Filter
    ffff8083e19a6ad0 - WFP Native MAC Layer LightWeight Filter
        Filter ffff8083e43df8f0, Miniport ffff8083e0f501a0 - Microsoft Kernel
        Debug Network Adapter
    ffff8083e19a6d70 - WFP 802.3 MAC Layer LightWeight Filter
```

```
Filter fffff8083e0d89c70, Miniport fffff8083e0f501a0 - Microsoft Kernel  
Debug Network Adapter
```

By using this filter handle we can now see more detailed information about it, such as its State, Higher filter handle, and Lower filter handle.

Console

```
3: kd> !ndiskd.filter fffff8083e14e8460
```

FILTER

Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000

```
Ndis handle      fffff8083e14e8460  
Filter driver    fffff8083e1a7fd90 - QoS Packet Scheduler  
Module context   fffff8083e26953e0  
Miniport         fffff8083e0f501a0 - Microsoft Kernel Debug Network  
Adapter
```

Network interface fffff8083e200f010

```
State            Running  
Datapath        Send only  
References      1  
Flags           RUNNING  
More flags      OID_TOP
```

```
Higher filter    fffff8083e0d89c70 - Microsoft Kernel Debug Network  
Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000  
Lower filter     fffff8083e43df8f0 - Microsoft Kernel Debug Network  
Adapter-WFP Native MAC Layer LightWeight Filter-0000
```

Driver handlers

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.filterdriver

Article • 10/25/2023

The **!ndiskd.filterdriver** extension displays information about an NDIS filter driver. If you run this extension with no parameters, !ndiskd will display a list of all filter drivers.

Console

```
!ndiskd.filterdriver -handle <xx> [-filters] [-handlers]
```

Parameters

-handle

Optional handle of an NDIS filter driver.

-filters

Displays instances of this driver's filters.

-handlers

Displays this driver's filter handlers.

DLL

Ndiskd.dll

Examples

Run **!ndiskd.filterdriver** with no parameters to see a list of all filter drivers on the system. In the following example, look for the Virtual WiFi Filter Driver, whose handle is fffffbc064cc83be0.

Console

```
0: kd> !ndiskd.filterdriver
fffffb064cccd4900 - QoS Packet Scheduler
fffffb064cc83be0 - Virtual WiFi Filter Driver
fffffb064cb91a10 - WFP vSwitch Layers LightWeight Filter
fffffb064cb8fd70 - WFP Native MAC Layer LightWeight Filter
fffffb064cb59b00 - WFP 802.3 MAC Layer LightWeight Filter
```

By clicking the filter driver handle from the previous example or by using it to enter the !ndiskd.filterdriver -handle command in the command window, you can get see more detailed information about that filter driver. In this case, for example, there are no filter modules for this filter driver.

```
Console

0: kd> !ndiskd.filterdriver fffffbc064cc83be0

FILTER DRIVER

Virtual WiFi Filter Driver

Ndis handle      fffffbc064cc83be0
Driver context    fffffbc064cc8e9d0
Ndis API version v6.50
Driver version   v1.0
Driver object     fffffbc064cc8e9d0
Driver image      vwififlt.sys

Bind flags        Optional, Modifying
Class             [Zero-length string]
References        1

FILTER MODULES

Filter module
[No filter modules were found]

HANDLERS

  Filter handler           Function pointer   Symbol (if
available)                                available)
  SetOptionsHandler        [None]
  SetFilterModuleOptionsHandler [None]
  AttachHandler            fffff80787d83b60  bp
  DetachHandler            fffff80787d84800  bp
  RestartHandler           fffff80787d86e20  bp
  PauseHandler             fffff80787d863e0  bp
  SendNetBufferListsHandler fffff80787d814d0  bp
  SendNetBufferListsCompleteHandler fffff80787d81940  bp
  CancelSendNetBufferListsHandler fffff80787d842f0  bp
  ReceiveNetBufferListsHandler fffff80787d817e0  bp
  ReturnNetBufferListsHandler fffff80787d81a80  bp
  OidRequestHandler        fffff80787d85ae0  bp
  OidRequestCompleteHandler fffff80787d85fd0  bp
  DirectOidRequestHandler fffff80787d84af0  bp
  DirectOidRequestCompleteHandler fffff80787d84e80  bp
  CancelDirectOidRequestHandler fffff80787d841d0  bp
  DevicePnPEventNotifyHandler fffff80787d849e0  bp
```

NetPnPEventHandler	fffff80787d85a40	bp
StatusHandler	fffff80787d877c0	bp

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.nbl

Article • 10/25/2023

The **!ndiskd.nbl** extension displays information about a **NET_BUFFER_LIST** (NBL) structure.

Console

```
!ndiskd.nbl [-handle <x>] [-basic] [-chain] [-info] [-data]
[-netmon] [-capfile <str>] [-launch] [-overwrite] [-log]
[-stacks] [-NblCurrentOwner]
```

Parameters

-handle

Required. Address of a **NET_BUFFER_LIST** structure.

-basic

Displays basic information about an NBL.

-chain

Displays all the NBLs and **NET_BUFFER**s in an NBL chain.

-info

Displays all the out-of-band information that is associated with an NBL.

-data

Displays the actual data payload of an NBL.

-netmon

Views the NBL chain in Microsoft Network Monitor.

-capfile

Specifies the path to which a netmon capture is saved.

-launch

Automatically launches netmon.exe after saving the capture file.

-overwrite

Allows overwriting the capture file if it already exists.

-log

Shows NBL log if NBL history logging is enabled.

-stacks

Includes callstacks with NBL log (use with *-log*).

-NblCurrentOwner

Shows the current owner of the NBL.

DLL

Ndiskd.dll

Examples

In the following example, NBL tracking has been enabled to extract a handle for an NBL from the NBL log. For more information about NBL tracking and the NBL log, see [!ndiskd.nbllog](#).

At the time of log collection, the NBL in this example was returned by the TCPIP6 protocol to the WFP Native Mac Layer LightWeight Filter.

```
Console

2: kd> !ndiskd.nbl fffffdf80149524a0
      NBL          fffffdf80149524a0    Next NBL        NULL
      First NB     fffffdf8014952610    Source
fffffdf80140c71a0 - Microsoft Kernel Debug Network Adapter
      Flags        INDICATED, RETURNED, NBL_ALLOCATED, PROTOCOL_020_0,
                    PROTOCOL_200_0

      Walk the NBL chain           Dump data payload
      Show out-of-band information
      Review NBL history
```

By clicking on the "Dump data payload" link from the previous example or by entering the **!ndiskd.nbl -handle -data** command, you can see the data payload of this NBL. In the following example, the NBL contains only one **NET_BUFFER** structure. To further explore the contents of that **NET_BUFFER** structure, run the [!ndiskd.nb -handle](#) command with its handle.

```
Console

2: kd> !ndiskd.nbl fffffdf80149524a0 -data
NET_BUFFER fffffdf8014952610
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER_LIST](#)

[NET_BUFFER](#)

[!ndiskd.nbllog](#)

[!ndiskd.nb](#)

!ndiskd.nb

Article • 10/25/2023

The **!ndiskd.nb** extension displays information about a [NET_BUFFER](#) (NB) structure.

Console

```
!ndiskd.nb [-handle <x>] [-verbosity <x>] [-basic] [-chain] [-data]
```

Parameters

-handle

Required. Address of a [NET_BUFFER](#) structure.

-verbosity

Level of detail to display.

-basic

Displays basic information about an NB.

-chain

Displays all the MDLs associated with an NB.

-data

Dumps the actual data payload of an NB.

DLL

Ndiskd.dll

Examples

The [NET_BUFFER](#) in the following examples was obtained from the [NET_BUFFER_LIST](#) in the Examples section of the [!ndiskd.nbl](#) topic. The NB's handle is fffffdf8014952610.

Console

```
2: kd> !ndiskd.nbl fffffdf80149524a0 -data
NET_BUFFER fffffdf8014952610
```

You can click the **NET_BUFFER**'s handle or run the **!ndiskd.nb -handle** command to see its details.

```
Console

2: kd> !ndiskd.nb fffffdf8014952610
    NB          fffffdf8014952610      Next NB          0
    Length       0                  Source pool
fffffdf80147e4a40
    First MDL    fffffdf8014a37930    DataOffset       0
    Current MDL [First MDL]        Current MDL offset 0

    View associated NBL
```

Use the **!ndiskd.nb -chain** command to see this **NET_BUFFER**'s MDL chain in addition to its basic details. In the following example, there is only one MDL. Its handle is **fffffdf8014a37930**.

```
Console

2: kd> !ndiskd.nb fffffdf8014952610 -chain
    NB          fffffdf8014952610      Next NB          0
    Length       0                  Source pool
fffffdf80147e4a40
    First MDL    fffffdf8014a37930    DataOffset       0
    Current MDL [First MDL]        Current MDL offset 0
    MDL [current]  fffffdf8014a37930  MDL Flags         c
    MappedSystemVa fffffdf8014bf0024  ByteCount        0n1514
    Process        [System process]  ByteOffset       0n36
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER](#)

[NET_BUFFER_LIST](#)

[!ndiskd.nbl](#)

!ndiskd.nblpool

Article • 10/25/2023

The **!ndiskd.nblpool** extension displays information about a [NET_BUFFER_LIST](#) (NBL) pool. If you run this extension with no parameters, !ndiskd will display a list of all allocated NBL pools in the system.

Console

```
!ndiskd.nblpool [-handle <x>] [-basic] [-allocations] [-find <str>] [-findnb  
<str>]  
[-findctx <str>] [-findctxtype <str>] [-findva <x>] [-findpa <x>]
```

Parameters

-handle

Handle of an NBL pool.

-basic

Displays basic information about the NBL pool.

-allocations

Displays all allocated NBLs.

-find

Filter the list of allocated NBLs using a debugger expression.

-findnb

Filter the list of allocated NBLs by linked [NET_BUFFER](#)s (NBs).

-findctx

Filter the list of allocated NBLs by context area.

-findctxtype

Override the datatype of the context area.

-findva

Find NBLs that contain an NB that straddles the given virtual address.

-findpa

Find NBLs that contain an NB that straddles the given physical address.

DLL

Ndiskd.dll

Examples

Enter the `!ndiskd.nblpool` command with no parameters to see a list of all allocated NBL pools. In this example, look for the NBL pool allocated by the kernel debugger network interface card (kdnic) with the KDNr Tag. Its handle is `fffffdf80147e4a40`.

```
Console

2: kd> !ndiskd.nblpool
      NBL Pool          Tag           Allocated by
      fffffdf80179b6a40  NiBP          WdNisDrv!CWFPLayer::Initialize+c6
      fffffdf8015ac6a40  EUNP
tunnel!TunnelEtherUdpGlobalInit+81
      fffffdf8015a78040  Nuio          ndisuo!ndisuoCreateBinding+15f
      fffffdf8015a77800  Nuio          ndisuo!ndisuoCreateBinding+13c
      fffffdf8015a63040  BaNB          rspndr!TopStartNetBufferModule+6d
      fffffdf8015a68a40  LLnb          mslldp!lldpProtSetOptions+49
      fffffdf8014654040  BaNB          lltdio!TopStartNetBufferModule+6d
      fffffdf801494ca40  Pcsb          pacer!PcFilterAttach+142
      fffffdf80147e4a40  KDNr          kdnic!NICAllocAdapter+178
      fffffdf80131ce040  bnvW          wfplwfs!DriverEntry+7a0
      fffffdf80139ffa40  Wfdp          wfplwfs!WfpRioInitialize+a4
      fffffdf8012061200  UNbl
NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
      fffffdf8013968a40  TcDN
NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
      fffffdf8013969a40  TNbl
NETIO!NetioAllocateNetBufferListNetBufferMdlAndDataPool+49
      fffffdf801397c040  StBn          NETIO!StreamPoolsInit+c1
      fffffdf8013088040  Wfra          NETIO!WfpNblInfoLibraryInit+b8
      fffffdf8012067440  Nnnn
NETIO!NetioInitializeNetBufferListLibrary+13e
      fffffdf8012067a40  Nnbl
NETIO!NetioInitializeNetBufferListLibrary+112
      fffffdf80131caa40  NDrt
ndis!ndisInitializePeriodicReceives+22f
      fffffdf80131d5a40  NDnd          ndis!DriverEntry+5e9
```

Click on the NBL pool's handle or enter the `!ndiskd.nblpool -handle` command to examine its details.

```
Console

2: kd> !ndiskd.nblpool fffffdf80147e4a40
```

NBL POOL

Ndis handle	fffffdf80147e4a40
Allocation tag	KDNr
Owner	
Allocated by	knic!NICAllocAdapter+178
Flags	CONTAINS_NET_BUFFER
Structure size	0n544
Context size	0
Data size	0

All allocated NBLs

To explore the NBLs contained in this NBL pool, click on the "All allocated NBLs" link at the bottom. Alternatively, you can also enter the **!ndiskd.nblpool -handle -allocations** command. As shown in the following example, this NBL pool contains more than 1024 NBLs so !ndiskd quit early. You can use the -force option to work around this limit and see all of the NBLs in this NBL pool.

Console

```
2: kd> !ndiskd.nblpool fffffdf80147e4a40 -allocations
```

ALL ALLOCATED NBLs

NBL	Active?
fffffdf8014951940	Allocated
fffffdf8014951b90	Allocated
fffffdf8014951de0	Allocated
fffffdf8014951030	Allocated
fffffdf80149524a0	Allocated
fffffdf80149526f0	Allocated
fffffdf8014952940	Allocated
fffffdf8014952b90	Allocated
fffffdf8014952de0	Allocated
fffffdf8014952030	Allocated
fffffdf80149534a0	Allocated
fffffdf80149536f0	Allocated
fffffdf8014953940	Allocated
fffffdf8014953b90	Allocated
fffffdf8014953de0	Allocated
fffffdf8014953030	Allocated
fffffdf80149544a0	Allocated
fffffdf80149546f0	Allocated
fffffdf8014954940	Allocated
...	
fffffdf80148b0b90	Allocated

```
fffffdf80148b0de0 Allocated
fffffdf80148b0030 Allocated
fffffdf80148b14a0 Allocated
fffffdf80148b16f0 Allocated
fffffdf80148b1940 Allocated
fffffdf80148b1b90 Allocated
fffffdf80148b1de0 Allocated
fffffdf80148b1030 Allocated
[Maximum of 1024 items read; quitting early. Rerun with the '-force'
option
to bypass this limit.]
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER_LIST](#)

[NET_BUFFER](#)

!ndiskd.nbpool

Article • 10/25/2023

The **!ndiskd.nbpool** extension displays information about a **NET_BUFFER** (NB) pool. If you run this extension with no parameters, !ndiskd will display a list of all allocated NB pools in the system.

Console

```
!ndiskd.nbpool [-handle <x>] [-allocations] [-find <str>] [-findva <x>] [-findpa <x>]
```

Parameters

-handle

Handle of an NB pool.

-allocations

Displays all allocated NBs.

-find

Filter the list of allocated NBs using a debugger expression.

-findva

Find NBs that straddle the given virtual address.

-findpa

Find NBs that straddle the given physical address.

DLL

Ndiskd.dll

Examples

Enter the **!ndiskd.nbpool** command with no parameters to see a list of all allocated NB pools. In this example, look for the NB pool allocated by the Netio service with the Nnbf Tag. Its handle is fffffdf801308ca40.

Console

```
2: kd> !ndiskd.nbpool
      NB Pool          Tag          Allocated by
      fffffdf8013963a40  UDNb
      NETIO!NetioAllocateNetBufferMdlAndDataPool+3c
      fffffdf801396aa40  TSNb
      NETIO!NetioAllocateNetBufferMdlAndDataPool+3c
      fffffdf801397d4c0  StBn        NETIO!StreamPoolsInit+90
      fffffdf801308ca40  Nnbf
      NETIO!NetioInitializeNetBufferListLibrary+dd
      fffffdf80131cba40  NDnd        ndis!DriverEntry+615
```

Click on the NB pool's handle or enter the **!ndiskd.nbpool -handle** command to examine its details.

Console

```
2: kd> !ndiskd.nbpool fffffdf801308ca40
```

NB POOL

```
Ndis handle      fffffdf801308ca40
Allocation tag    Nnbf
Owner
Allocated by     NETIO!NetioInitializeNetBufferListLibrary+dd

Flags            [No flags set]
Structure size   0n176
Data size        0
```

[All allocated NBs](#)

To explore the NBs contained in this NB pool, click on the "All allocated NBs" link at the bottom. Alternatively, you can also enter the **!ndiskd.nbpool -handle -allocations** command. As shown in the following example, this NB pool contains more than 1024 NBs so !ndiskd quit early. You can use the -force option to work around this limit and see all of the NBs in this NB pool.

Console

```
2: kd> !ndiskd.nbpool fffffdf801308ca40 -allocations
```

ALL ALLOCATED NBs

NB	Active?
fffffdf8016ea4360	Allocated
fffffdf801744df50	Allocated
fffffdf8016932860	Allocated

```
fffffdf8016e31500 Allocated
fffffdf80174eade0 Allocated
fffffdf8017daa900 Allocated
fffffdf8017c8c680 Allocated
fffffdf80166b23b0 Allocated
fffffdf80164fea70 Allocated
fffffdf8012845990 Allocated
fffffdf8017d692d0 Allocated
fffffdf8017cdc090 Allocated
fffffdf8012771780 Allocated
fffffdf80158a3550 Allocated
fffffdf8012eef5c0 Allocated
fffffdf80127719d0 Allocated
fffffdf8015119570 Allocated
fffffdf8012e18d40 Allocated
fffffdf8017929b10 Allocated
fffffdf8016d4e430 Allocated

...
fffffdf8015ffb0 Allocated
fffffdf8015ec1b10 Freed
fffffdf80158e56d0 Allocated
fffffdf8016272110 Freed
fffffdf8015d8e030 Freed
fffffdf8015d8e770 Freed
fffffdf80158ddc30 Freed
fffffdf801584acc0 Freed
fffffdf8015846b40 Freed
fffffdf8015a06c50 Freed
fffffdf801480c300 Freed
fffffdf8015e48f50 Freed
fffffdf8015de64e0 Freed
fffffdf8015ddff50 Freed
[Maximum of 1024 items read; quitting early. Rerun with the '-force'
option
to bypass this limit.]
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER](#)

!ndiskd.pendingnbls

Article • 10/25/2023

The **!ndiskd.pendingnbls** extension displays pending NBLs ([NET_BUFFER_LISTs](#)) that are in transit.

Console

```
!ndiskd.pendingnbls [-handle <x>] [-fullstack] [-verbosity <x>]
```

Parameters

-handle

Handle of an NDIS miniport, filter, or open.

-fullstack

Shows pending NBLs from the entire stack associated with the handle.

-verbosity

Level of detail to display.

DLL

Ndiskd.dll

Examples

!ndiskd.pendingnbls can be passed the handle of an NDIS miniport, filter, or open. The following series of examples use a miniport handle. To see a list of all miniports and their associated minidrivers, run the **!ndiskd.netadapter** extension with no parameters. In the following example output, look for the Microsoft Kernel Debug Network Adapter, whose handle is fffffe00bc3f701a0. Its minidriver's handle is fffffe00bc51b9ae0.

Console

```
0: kd> !ndiskd.netadapter
  Driver          NetAdapter        Name
  fffffe00bc6e12ae0  fffffe00bc6e4e1a0  Microsoft ISATAP Adapter #2
  fffffe00bc51b9ae0  fffffe00bc3f701a0  Microsoft Kernel Debug Network
                           Adapter
```

To see the pending NBLs for a miniport, set a breakpoint on its minidriver's SendNetBufferListsHandler. Use the minidriver's handle to run the [!ndiskd.minidriver -handle -handlers](#) command to see a list of its handlers, then click the "bp" link to the right of the SendNetBufferListsHandler. You can alternatively enter the [bp -handle](#) command in the command line.

```
Console

0: kd> !ndiskd.minidriver fffffe00bc51b9ae0 -handlers

HANDLERS

    NDIS Handler                                Function pointer   Symbol (if
available)
    InitializeHandlerEx                         fffff80ae9618230 bp
    SetOptionsHandler                           fffff80ae9612800 bp
    HaltHandlerEx                             fffff80ae9618040 bp
    ShutdownHandlerEx                         fffff80ae96122c0 bp

    CheckForHangHandlerEx                     fffff80ae9612810 bp
    ResetHandlerEx                            fffff80ae9612f70 bp

    PauseHandler                             fffff80ae9618000 bp
    RestartHandler                           fffff80ae9618940 bp

    OidRequestHandler                        fffff80ae9611c90 bp
    CancelOidRequestHandler                  fffff80ae96122c0 bp
    DirectOidRequestHandler                 [None]
    CancelDirectOidRequestHandler            [None]
    DevicePnPEventNotifyHandler             fffff80ae96189a0 bp

    SendNetBufferListsHandler                fffff80ae9611870 bp
    ReturnNetBufferListsHandler              fffff80ae9611b50 bp
    CancelSendHandler                        fffff80ae96122c0 bp
```

After setting the breakpoint on the SendNetBufferListsHandler, enter the **g** command to let the debuggee target machine run and hit the breakpoint.

```
Console

0: kd> bp fffff80ae9611870
0: kd> g
Breakpoint 0 hit
fffff80a`e9611870 4053          push    rbx
```

Now, after hitting the minidriver's SendNetBufferListsHandler breakpoint, you can see any pending NBLs for the miniport by entering the [!ndiskd.pendingnbls -handle](#) command with the miniport's handle.

Note The debuggee target machine in this example was loading a web page when it hit the breakpoint, so traffic was flowing through the miniport's datapath. Therefore, it had a pending NBL to send. Even after setting a breakpoint on one or more of the NBL handlers for the minidriver, you may not see any pending NBLs if there were no activity in the datapath.

```
Console

0: kd> !ndiskd.pendingnbls fffffe00bc3f701a0

PHASE 1/3: Found 20 NBL pool(s).
PHASE 2/3: Found 342 freed NBL(s).

    Pending Nbl          Currently held by
    fffffe00bc5545c60    fffffe00bc3f701a0 - Microsoft Kernel Debug Network
Adapter [NetAdapter]

PHASE 3/3: Found 1 pending NBL(s) of 4817 total NBL(s).
Search complete.
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER_LIST](#)

[!ndiskd.netadapter](#)

[!ndiskd.minidriver](#)

[bp, bu, bm \(Set Breakpoint\)](#)

!ndiskd.nbllog

Article • 10/25/2023

The `!ndiskd.nbllog` extension displays the log of all NBL ([NET_BUFFER_LIST](#)) activity on the system.

Console

```
!ndiskd.nbllog [-stacks]
```

Parameters

`-stacks`

Include callstacks.

DLL

Ndiskd.dll

Remarks

Important `!ndiskd.nbllog` requires NBL tracking to be enabled on the debuggee target machine. NBL tracking is not enabled by default in all configurations of Windows. If NBL tracking is not enabled, !ndiskd will give you instructions on how to enable it, as shown in the following snippet.

Console

```
0: kd> !ndiskd.nbllog
This command requires NBL tracking to be enabled on the debuggee target
machine. (By default, client operating systems have level 1, and
servers
have level 0). To enable, set this REG_DWORD value to a nonzero value
on
the target machine and reboot the target machine:

HKLM\SYSTEM\CurrentControlSet\Services\NDIS\Parameters ! TrackNblOwner
Possible Values (features are cumulative)
* 0: Disable all tracking.
* 1: Track the most recent owner of each NBL (enables
!ndiskd.pendingnbls)
* 2: Scan for leaks at runtime (use with StuckNblReaction)
* 3: Keep a full history of all activity (enables !ndiskd.nbl -log)
```

```
* 4: Take stack capture snapshots (enables !ndiskd.nbl -log -stacks)
This command requires level 3 or higher.
```

The NBL log shows network traffic on the system. [!ndiskd.netreport](#) parses the NBL tracking log to display this network traffic visually. Therefore, if NBL tracking is not enabled, [!ndiskd.netreport](#) will not be able to show you this information.

Examples

After you have enabled NBL tracking on the target debuggee machine, enter the [!ndiskd.nbllog](#) command to see the log of all NBL traffic on the system. As shown in the example below, running [!ndiskd.nbllog](#) with no parameters will limit output to 200 events, which can be bypassed by rerunning the command with the *-force* option. The middle of the output in this example has been excised for brevity.

Console

```
0: kd> !ndiskd.nbllog
      NBLs          Processor        Event           Detail
      fffffe00bc71453f0    CPU  0            Freed
      fffffe00bc7163b40    CPU  2            Allocated
      fffffe00bc7163b40    CPU  2            ProtocolSent
fffffe00bc5ac4880 - QoS Packet Scheduler-0000
      fffffe00bc7163b40    CPU  2            FilterSent
fffffe00bc5ac5c70 - WFP Native MAC Layer LightWeight Filter-0000
      fffffe00bc7163b40    CPU  2, IRQL=DPC  FilterSent
fffffe00bc3f701a0 - Microsoft Kernel Debug Network Adapter
      fffffe00bc7163b40    CPU  2, IRQL=DPC  SentToMiniport
fffffe00bc3f701a0 - Microsoft Kernel Debug Network Adapter
      fffffe00bc7163b40    CPU  0, IRQL=DPC  MiniportSendCompleted
fffffe00bc5ac5c70 - WFP Native MAC Layer LightWeight Filter-0000
      fffffe00bc7163b40    CPU  0, IRQL=DPC  FilterSendCompleted
fffffe00bc5ac4880 - QoS Packet Scheduler-0000
      fffffe00bc7163b40    CPU  0, IRQL=DPC  FilterSendCompleted send complete
in NDIS, sorting to Opens
      fffffe00bc7163b40    CPU  0, IRQL=DPC  SendCompleted
fffffe00bc5ab7c10 - TCPIP6

...
      fffffe00bc6b469b0    CPU  2            Allocated
      fffffe00bc6b469b0    CPU  2            Freed
      fffffe00bc64a3690    CPU  2            Allocated
      fffffe00bc64a3690    CPU  2            ProtocolSent
fffffe00bc5ac4880 - QoS Packet Scheduler-0000
      fffffe00bc64a3690    CPU  2            FilterSent
fffffe00bc5ac5c70 - WFP Native MAC Layer LightWeight Filter-0000
      fffffe00bc64a3690    CPU  2, IRQL=DPC  FilterSent
```

```
fffffe00bc3f701a0 - Microsoft Kernel Debug Network Adapter
    fffffe00bc64a3690   CPU 2, IRQL=DPC   SentToMiniport
fffffe00bc3f701a0 - Microsoft Kernel Debug Network Adapter
    fffffe00bc3cf2d10   CPU 1           Allocated
    fffffe00bc7bc6030   CPU 1           Allocated
    fffffe00bc3cf2d10   CPU 1           ProtocolSent
fffffe00bc5ac4880 - QoS Packet Scheduler-0000

Maximum of 200 events printed; quitting early.
Rerun with the '-force' option to bypass this limit.
```

For a more detailed description of how to interpret the results of `!ndiskd.nbllog`, see [!ndiskd.nbl -log](#) on the NDIS blog.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NET_BUFFER_LIST](#)

[!ndiskd.nbl -log](#)

!ndiskd.oid

Article • 10/25/2023

The **!ndiskd.oid** extension displays information about an NDIS OID request. If you run this extension with no parameters, !ndiskd will display a list of all pending OID requests on all miniports and filters. Each miniport or filter has at most one pending OID request and any number of queued OID requests.

Note that filters typically clone OID requests and pass the clone down. This means that even if a protocol issues a single OID request, there may be multiple instances of cloned requests: one in each filter and another in the miniport. **!ndiskd.oid** will show each clone separately, so you may see more pending OIDs than the protocol has actually issued.

Console

```
!ndiskd.oid [-handle <x>] [-legacyoid] [-nolimit] [-miniport <x>]
```

Parameters

-handle

Handle of an NDIS_OID_REQUEST

-legacyoid

Treats as a legacy NDIS_REQUEST instead of an NDIS_OID_REQUEST.

-nolimit

Does not limit the number of pending OIDs that are displayed.

-miniport

Finds pending OID requests on this miniport's stack.

DLL

Ndiskd.dll

Remarks

!ndiskd.oid shows you a list of all the pending OIDs on the system at a time, so it can be helpful in debugging system hangs or [0x9F bug check](#) situations (DRIVER_POWER_STATE_FAILURE). For example, suppose analyzing a fictitious 0x9F bug check revealed that the system was hung on an IRP and was waiting for NDIS. In NDIS,

IRPs from the OS are translated into OIDs, including power transitions, so by running **!ndiskd.oid** you could see that, in this example, a device at the bottom of the stack might have been clinging to an [OID_PNP_SET_POWER](#) and hung the rest of the stack. NDIS drivers should not pend an OID for more than one second, so you could then investigate why that device kept the OID pending for too long to try to solve the issue.

Examples

To see an example of pending OIDS on a system that is running normally, set a breakpoint on a miniport's OID request handler routine (in the miniport's corresponding miniport driver). First, run the **!ndiskd.minidriver** command with no parameters to get a list of miniport drivers on the system. In this example output, look for the handle for the kdnic minidriver, fffffdf801418d650..

```
Console

3: kd> !ndiskd.minidriver
fffffdf8015a98380 - tunnel
fffffdf801418d650 - kdnic
```

Click on the handle for the minidriver, then click on the "Handlers" link at the bottom of its details page to see the list of its handlers. You can alternatively enter the **!ndiskd.minidriver -handle -handlers** command. Once you have the list of the minidriver's handlers, look for the OidRequestHandler, whose handle is fffff80f1fd71c90 in this example.

```
Console

2: kd> !ndiskd.minidriver fffffdf801418d650 -handlers

HANDLERS

    NDIS Handler                                Function pointer   Symbol (if
available)
    InitializeHandlerEx                         fffff80f1fd78230  bp
    SetOptionsHandler                           fffff80f1fd72800  bp
    HaltHandlerEx                             fffff80f1fd78040  bp
    ShutdownHandlerEx                          fffff80f1fd722c0  bp

    CheckForHangHandlerEx                     fffff80f1fd72810  bp
    ResetHandlerEx                            fffff80f1fd72f70  bp

    PauseHandler                             fffff80f1fd78000  bp
    RestartHandler                           fffff80f1fd78940  bp

    OidRequestHandler                        fffff80f1fd71c90  bp
```

CancelOidRequestHandler	fffff80f1fd722c0	bp
DirectOidRequestHandler	[None]	
CancelDirectOidRequestHandler	[None]	
DevicePnPEventNotifyHandler	fffff80f1fd789a0	bp
SendNetBufferListsHandler	fffff80f1fd71870	bp
ReturnNetBufferListsHandler	fffff80f1fd71b50	bp
CancelSendHandler	fffff80f1fd722c0	bp

Now either click on the "bp" link to the right of the OidRequestHandler or enter the [bp - handle](#) command with its handle to set a breakpoint on that routine. Next, type the `g` command to allow your debuggee target machine to run and hit the breakpoint you just set.

Console

```
2: kd> bp fffff80f1fd71c90
2: kd> g
Breakpoint 1 hit
fffff80f`1fd71c90 448b4204      mov     r8d,dword ptr [rdx+4]
```

Once you have triggered the breakpoint on a minidriver's OID request handler routine as shown by the previous example, you can run the `!ndiskd.oid` command to see a list of all the pending OIDs on the system.

Console

```
1: kd> !ndiskd.oid

ALL PENDING OIDS

NetAdapter          fffffdf80140c71a0 - Microsoft Kernel Debug Network
Adapter
    Current OID      OID_GEN_STATISTICS
    Filter           fffffdf8014950c70 - Microsoft Kernel Debug Network
Adapter-WFP Native MAC Layer LightWeight Filter-0000
    Current OID      OID_GEN_STATISTICS
    Filter           fffffdf801494dc70 - Microsoft Kernel Debug Network
Adapter-QoS Packet Scheduler-0000
    Current OID      OID_GEN_STATISTICS
```

In this example, the OID pending is [OID_GEN_STATISTICS](#). When you look at the results of `!ndiskd.oid`, recall that filters clone OID requests and pass them down the stack, and OIDs typically get passed from filter to filter to miniport. Therefore, although it may look like there are three separate OID requests with the same name in this example, there is

actually one logical operation taking place which was physically spread across 3 OIDs and on 3 drivers.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[0x9F bug check](#)

[OID_PNP_SET_POWER](#)

[bp, bu, bm \(Set Breakpoint\)](#)

[OID_GEN_STATISTICS](#)

[NDIS OIDs](#)

[NDIS OID Request Interface](#)

!ndiskd.interfaces

Article • 10/25/2023

The **!ndiskd.interfaces** extension displays information about a network interface. If you run this extension with no parameters, !ndiskd will display a list of all network interfaces.

For more information about network interfaces, see [NDIS Network Interfaces](#).

Console

```
!ndiskd.interfaces -handle <x> [-luid <x>]
```

Parameters

-handle

Optional handle of a network interface.

-luid

[NetLuid](#) (Net Locally Unique Identifier) of a network interface.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.interfaces** extension with no parameters to see a list of all network interfaces on the system. In this example, look for the Intel(R) 82579LM Gigabit Network Connection interface. Its handle is fffffdf80139f8a20.

Console

```
1: kd> !ndiskd.interfaces
      Interface
      fffffdf801494fa20 - Microsoft Kernel Debug Network Adapter-WFP Native MAC
      Layer LightWeight Filter-0000
      fffffdf801494c010 - Microsoft Kernel Debug Network Adapter-QoS Packet
      Scheduler-0000
      fffffdf801494ba20 - Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC
      Layer LightWeight Filter-0000
      fffffdf80139b3a20 - Microsoft Kernel Debug Network Adapter
      fffffdf80139f8a20 - Intel(R) 82579LM Gigabit Network Connection
      fffffdf80139baa20 - WAN Miniport (IP)
```

```
fffffdf80139a4a20 - WAN Miniport (IPv6)
fffffdf80131d0010 - WAN Miniport (Network Monitor)
fffffdf80131cda20 - WAN Miniport (PPPOE)
fffffdf80139b6a20 - Software Loopback Interface 1
fffffdf80139b0a20 - Microsoft ISATAP Adapter
fffffdf80139ada20 - WAN Miniport (SSTP)
fffffdf80131cf010 - WAN Miniport (IKEv2)
fffffdf80139fea20 - WAN Miniport (L2TP)
fffffdf80139a7a20 - WAN Miniport (PPTP)
fffffdf80139aaa20 - Microsoft ISATAP Adapter #2
fffffdf80139fba20 - Teredo Tunneling Pseudo-Interface
```

By clicking on the handle for an interface or by entering the !ndiskd.interfaces -handle command, you can see the details about that interface, including its Identifier information and its current state. In this example, you can see that the Intel(R) 82579LM Gigabit Network Connection is an Ethernet connection (its ifAlias) and is in a MediaConnectUnknown state for its connection (as it has been reserved for use by the Windows kernel debugger).

Console

```
1: kd> !ndiskd.interfaces fffffdf80139f8a20
```

INTERFACE

Ethernet

```
Ndis handle      fffffdf80139f8a20
IfProvider       fffffdf80131ca8d0 - The NDIS interface provider
Provider context fffffdf80139f8a20

ifType          IF_TYPE_ETHERNET_CSMACD
Media type      802.3
Physical medium 802.3
Access type     BROADCAST
Direction type   SEND_AND_RECEIVE
Connection type  DEDICATED
```

```
ifConnectorPresent No
```

```
Network          fffffdf80139b8900 - [Unnamed network]
Compartment      fffffdf80139b9940 - Compartment #1
```

IDENTIFIERS

```
ifAlias          Ethernet
ifDescr          Intel(R) 82579LM Gigabit Network Connection
ifName (NET_LUID) 06:8001
ifPhysAddress    18-03-73-c1-e8-72
```

ifIndex	0n14
ifGuid	cbddfde1-5570-4c65-9d47-52d63abce00c

STATE

Connected	MediaConnectUnknown
ifOperStatus	NOT_PRESENT
Link speed	0 [Speed is not applicable]
ifMtu	0
Duplex	UnknownDuplex

Refer to RFC 2863 for definitions of many of these terms

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NDIS Network Interfaces](#)

[NET_LUID Value](#)

!ndiskd.ifprovider

Article • 10/25/2023

The **!ndiskd.ifprovider** extension displays information about an [NDIS interface provider](#) (IfProvider). If you run this extension with no parameters, !ndiskd will display a list of all registered NDIS interface providers.

Console

```
!ndiskd.ifprovider [-handle <x>]
```

Parameters

-handle

Optional handle of an IfProvider.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ifprovider** extension with no parameters to get a list of all registered IfProviders.

Console

```
1: kd> !ndiskd.ifprovider
IfProvider
fffffd20d14334180 - wanarp
fffffd20d1264a950 - wfplwfs
fffffd20d11deae00 - The NDIS loopback provider
fffffd20d11deae70 - The NDIS interface provider
```

You can see from the previous example that the debuggee machine has four interface providers registered. Two of them are NDIS interface providers.

Note Interface providers are a generic concept and aren't required to be miniport drivers. While a miniport driver may choose to register as an interface provider if desired, most miniport drivers do not do so because NDIS has a built-in interface provider. The NDIS built-in interface provider automatically provides interfaces for every

miniport driver, every Light-Weight Filter (LWF) module, and the loopback interface. For more information, see [NDIS interface provider](#).

The following example shows the details for the "wanarp" interface provider in the previous example, whose handle is fffffd20d14334180.

```
Console

1: kd> !ndiskd.ifprovider fffffd20d14334180

IF PROVIDER

wanarp
Ndis handle      fffffd20d14334180

INTERFACES

Interface
[No interfaces found]

HANDLERS

Protocol handler          Function pointer   Symbol (if
available)                available
QueryObjectHandler        ffffff80d2f0414b0  bp
wanarp!WanNdisIfQueryHandler
SetObjectHandler          ffffff80d2f04bd10  bp
wanarp!WanNdisIfSetHandler
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Registering as an Interface Provider](#)

!ndiskd.ifstacktable

Article • 10/25/2023

The **!ndiskd.ifstacktable** extension displays the network interface stack table (ifStackTable).

For more information about the interface stack table, see [Maintaining a Network Interface Stack](#).

```
Console
!
!ndiskd.ifstacktable
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ifstacktable** command to see the ifStackTable.

```
Console
3: kd> !ndiskd.ifstacktable

INTERFACE STACK TABLE

  Lower interface      Lower IfIndex      Higher IfIndex      Higher
  interface
  fffffdf80139b3a20    6                  15
  fffffdf801494fa20
  fffffdf801494fa20    15                  16
  fffffdf801494c010
  fffffdf801494c010    16                  17
  fffffdf801494ba20
```

NDIS maintains the stack table for NDIS miniport adapters, NDIS 5.x filter intermediate drivers, and NDIS filter modules, whereas [NDIS MUX Intermediate Drivers](#) drivers are

required to specify the internal interface relationship between the virtual miniport interface and the protocol lower interface. Therefore, the ifStackTable could be useful for seeing the interface stack relationships in a system with more complicated MUX drivers installed.

Since there are no NDIS MUX Intermediate drivers installed on this example system, the ifStackTable only shows the stack relationships that NDIS has provided. In the following example, clicking on the handle for the Lower interface of the third row (handle fffffdf801494c010, Lower IfIndex 16) shows the interface for the QoS Packet Scheduler.

```
Console

3: kd> !ndiskd.interface fffffdf801494c010

INTERFACE

[Zero-length string]

Ndis handle      fffffdf801494c010
IfProvider        fffffdf80131ca8d0 - The NDIS interface provider
NDIS filter       fffffdf801494dc70 - Microsoft Kernel Debug Network
Adapter-QoS Packet Scheduler-0000

ifType           IF_TYPE_ETHERNET_CSMACD
Media type       802.3
Physical medium NdisPhysicalMediumOther
Access type      BROADCAST
Direction type   SEND_AND_RECEIVE
Connection type  DEDICATED

ifConnectorPresent No

Network          fffffdf80139b8900 - [Unnamed network]
Compartment      fffffdf80139b9940 - Compartment #1

IDENTIFIERS

ifAlias          [Zero-length string]
ifDescr          Microsoft Kernel Debug Network Adapter-QoS Packet
Scheduler-0000
ifName (NET_LUID) 06:01
ifPhysAddress    18-03-73-c1-e8-72

ifIndex          0n16
ifGuid           fc2a0ae1-b103-11e6-b724-806e6f6e6963

STATE

Connected      Connected
```

```

ifOperStatus      DORMANT
ifOperStatusFlags DORMANT_PAUSED

Link speed        1000000000 (1 Gbps)
ifMtu             0x1500
Duplex            FullDuplex

```

Refer to RFC 2863 for definitions of many of these terms

Continuing the same example, clicking the handle for the Higher interface of the third row (handle fffffdf801494ba20, Higher IfIndex 17) shows the interface for the WFP 802.3 MAC Layer LightWeight Filter.

Console

```
3: kd> !ndiskd.interface fffffdf801494ba20
```

INTERFACE

[Zero-length string]

```

Ndis handle          fffffdf801494ba20      [type it]
IfProvider           fffffdf80131ca8d0 - The NDIS interface provider
NDIS filter          fffffdf801494c670 - Microsoft Kernel Debug Network
Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000

ifType               IF_TYPE_ETHERNET_CSMACD
Media type           802.3
Physical medium     NdisPhysicalMediumOther
Access type          BROADCAST
Direction type       SEND_AND_RECEIVE
Connection type      DEDICATED

```

ifConnectorPresent No

```

Network              fffffdf80139b8900 - [Unnamed network]
Compartment          fffffdf80139b9940 - Compartment #1

```

IDENTIFIERS

```

ifAlias              [Zero-length string]
ifDescr              Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC
Layer LightWeight Filter-0000
ifName (NET_LUID)   06:02
ifPhysAddress        18-03-73-c1-e8-72

ifIndex              0x17
ifGuid               fc2a0ae0-b103-11e6-b724-806e6f6e6963

```

STATE

Connected	Connected
ifOperStatus	DORMANT
ifOperStatusFlags	DORMANT_PAUSED
Link speed	1000000000 (1 Gbps)
ifMtu	0x1500
Duplex	FullDuplex

Refer to RFC 2863 for definitions of many of these terms

This shows that the WFP 802.3 MAC Layer LightWeight Filter sits above the QoS Packet Scheduler filter in the network interface stack. You can confirm this by running the [!ndiskd.netreport](#) extension, which shows you the network stack visually.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Maintaining a Network Interface Stack](#)

[NDIS MUX Intermediate Drivers](#)

[!ndiskd.netreport](#)

!ndiskd.compartments

Article • 10/25/2023

The **!ndiskd.compartments** extension displays all network compartments.

```
Console  
!ndiskd.compartments
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Remarks

Compartments are a way that NDIS manages interfaces. Third party interface providers only use the primary compartment, as described in the **CompartmentId** member of the [NDIS_BIND_PARAMETERS](#) structure.

Examples

Run the **!ndiskd.compartments** extension to see a list of all network compartments. In this example, there is only one compartment (the primary one).

```
Console  
3: kd> !ndiskd.compartments  
Compartment      fffffdf80139b9940  
ID              1  
Loopback Network   fffffdf80139b8900  
Loopback Interface fffffdf80139b6a20  
Networks:  
          fffffdf80139b8900      [Unnamed network]
```

See also

Network Driver Design Guide

Windows Vista and Later Networking Reference

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

NDIS_BIND_PARAMETERS

!ndiskd.pkt

Article • 10/25/2023

Warning This extension is for legacy NDIS 5.x drivers. The [NDIS_PACKET](#) structure and its associated architecture have been deprecated.

The **!ndiskd.pkt** extension displays information about an [NDIS_PACKET](#) structure.

Console

```
!ndiskd.pkt [-packet] [-verbosity]
```

Parameters

Packet

Specifies the address of the packet.

Verbosity

Specifies the amount of detail to be displayed.

DLL

Ndiskd.dll

See also

[NDIS_PACKET](#)

!ndiskd.pktpools

Article • 10/25/2023

Warning This extension is for legacy NDIS 5.x drivers. The [NDIS_PACKET](#) structure and its associated architecture have been deprecated.

The **!ndiskd.pktpools** extension displays a list of all allocated packet pools.

Console

```
!ndiskd.pktpools
```

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.pktpools** extension to see a list of all allocated packet pools on the system. Note that the handles for the packet pools are not clickable, which means you can't explore further information about the packet pool. This is because NDIS does not use packet pools starting with NDIS 6.0, so these pools are allocated only for legacy drivers which may still be on older systems. The debuggee machine in this example does not have any legacy NDIS 5.x drivers installed so the packet pools are not used. This example is for illustrative purposes only.

Console

```
3: kd> !ndiskd.pktpools
Pool      Allocator  BlocksAllocated  BlockSize  PktsPerBlock  PacketLength
fffffdf80131d58c0  ffffff80f1fbe3e8f  0x1        0x1000       0xa
0x190    ndis!DriverEntry+6af
fffffdf80131d5940  ffffff80f1fbe3e71  0x1        0x1000       0xa
0x180    ndis!DriverEntry+691
```

See also

[Windows 2000 and Windows XP Networking Design Guide](#)

[Windows 2000 and Windows XP Networking Reference](#)

`NDIS_PACKET`

!ndiskd.findpacket

Article • 10/25/2023

Warning This extension is for legacy NDIS 5.x drivers. The [NDIS_PACKET](#) structure and its associated architecture have been deprecated.

The **!ndiskd.findpacket** extension finds the specified packets.

Console

```
!ndiskd.findpacket [-VirtualAddress] [-PoolAddress]
```

Parameters

VirtualAddress

Specifies a virtual address that is contained in the desired packet.

PoolAddress

Specifies a pool address. All unreturned packets in this pool will be displayed.

DLL

Ndiskd.dll

See also

[NDIS_PACKET](#)

!ndiskd.vc

Article • 10/25/2023

The **!ndiskd.vc** extension displays a Connection-Oriented (CoNDIS) virtual connection (VC).

Console

```
!ndiskd.vc -handle <xx>
```

Parameters

-handle

Required. Handle of a VC pointer.

DLL

Ndiskd.dll

Remarks

For more information about CoNDIS, see [Connection-Oriented NDIS](#).

For more information about CoNDIS virtual connections, see [Virtual Connections](#).

Examples

CoNDIS is used in certain situations such as connecting to a VPN, so running **!ndiskd.vc** will not show you results unless a miniport driver on your system has created and activated a CoNDIS virtual connection. The following example shows results from a machine that is connected to a VPN network. First, run the [!ndiskd.netadapter](#) extension with no parameters to see a list of miniports and miniport drivers on the system. In the following output, look for the miniport driver for the Marvell AVASTAR Wireless-AC Network Controller network adapter. Its handle is fffffc804af2e3710.

Console

```
1: kd> !ndiskd.netadapter
      Driver          NetAdapter        Name
      fffffc804af2e3710  fffffc804b9e6f1a0  Marvell AVASTAR Wireless-AC
      Network Controller
```

fffffc804b99b9020	fffffc804b9c301a0	WAN Miniport (Network Monitor)
fffffc804b99b9020	fffffc804b9c2a1a0	WAN Miniport (IPv6)
fffffc804b99b9020	fffffc804b8a8a1a0	WAN Miniport (IP)
fffffc804ae9d7020	fffffc804b9ceb1a0	WAN Miniport (PPPOE)
fffffc804b9ca5900	fffffc804b9e601a0	WAN Miniport (PPTP)
fffffc804b99dc720	fffffc804b99b01a0	WAN Miniport (L2TP)
fffffc804b86581b0	fffffc804b9c6c1a0	WAN Miniport (IKEv2)
fffffc804ad4a7250	fffffc804b99651a0	WAN Miniport (SSTP)
fffffc804b11c4020	fffffc804b85821a0	Microsoft ISATAP Adapter
fffffc804b11c4020	fffffc804b71731a0	Microsoft ISATAP Adapter #2
fffffc804ad725020	fffffc804b05e71a0	Surface Ethernet Adapter #2
fffffc804b0bf0020	fffffc804b0c011a0	Bluetooth Device (Personal Area Network)
fffffc804aef695e0	fffffc804aed331a0	TAP-Windows Adapter V9

Next, enter the `!ndiskd.vc` command with the miniport driver's handle to see the virtual connections opened by that miniport driver.

Console

```
1: kd> !ndiskd.vc fffffc804af2e3710

VIRTUAL CALL

[Zero-length string]
Ndis handle      fffffc804af2e3710
VC Index         0

AF                ffffff80965fd5888
Call Flags        [No flags set]
VC Flags          [Unrecognized flags 04a80100] VC_ACTIVATE_PENDING

Miniport          ffffff80965ffaa20 - [Invalid NetAdapter]
Miniport Context ffffff80965ffaad0

Call Manager      ffffff80965ff9b50 - [Invalid Open]
Call Manager Context ffffff80965ff9c70

Client            fffffc804af96fd78 - [Invalid Open]
Client Context   00003206
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[**!ndiskd.help**](#)

[Connection-Oriented NDIS](#)

[Virtual Connections](#)

[**!ndiskd.netadapter**](#)

!ndiskd.af

Article • 10/25/2023

The **!ndiskd.af** extension displays a Connection-Oriented NDIS (CoNDIS) address family (AF).

Console

```
!ndiskd.af -handle <xx>
```

Parameters

-handle

Required. Handle of a CoNDIS address family.

DLL

Ndiskd.dll

Remarks

For more information about CoNDIS, see [Connection-Oriented NDIS](#).

For more information about CoNDIS address families, see [Address Families](#).

Examples

CoNDIS is used in certain situations such as connecting to a VPN, so running **!ndiskd.af** will not show you results unless a miniport driver on your system has created and activated a CoNDIS virtual connection. The following example shows results from a machine that is connected to a VPN network. First, run the [!ndiskd.netadapter](#) extension with no parameters to see a list of miniports and miniport drivers on the system. In the following output, look for the miniport driver for the Marvell AVASTAR Wireless-AC Network Controller network adapter. Its handle is fffffc804af2e3710.

Console

```
1: kd> !ndiskd.netadapter
      Driver          NetAdapter        Name
      fffffc804af2e3710  fffffc804b9e6f1a0  Marvell AVASTAR Wireless-AC
      Network Controller
```

fffffc804b99b9020	fffffc804b9c301a0	WAN Miniport (Network Monitor)
fffffc804b99b9020	fffffc804b9c2a1a0	WAN Miniport (IPv6)
fffffc804b99b9020	fffffc804b8a8a1a0	WAN Miniport (IP)
fffffc804ae9d7020	fffffc804b9ceb1a0	WAN Miniport (PPPOE)
fffffc804b9ca5900	fffffc804b9e601a0	WAN Miniport (PPTP)
fffffc804b99dc720	fffffc804b99b01a0	WAN Miniport (L2TP)
fffffc804b86581b0	fffffc804b9c6c1a0	WAN Miniport (IKEv2)
fffffc804ad4a7250	fffffc804b99651a0	WAN Miniport (SSTP)
fffffc804b11c4020	fffffc804b85821a0	Microsoft ISATAP Adapter
fffffc804b11c4020	fffffc804b71731a0	Microsoft ISATAP Adapter #2
fffffc804ad725020	fffffc804b05e71a0	Surface Ethernet Adapter #2
fffffc804b0bf0020	fffffc804b0c011a0	Bluetooth Device (Personal Area Network)
fffffc804aef695e0	fffffc804aed331a0	TAP-Windows Adapter V9

Next, enter the !ndiskd.af command with the miniport driver's handle to see the address family for this miniport driver, which is acting as a connection-oriented client.

Console

```
1: kd> !ndiskd.af fffffc804af2e3710
```

ADDRESS FAMILY

Ndis handle	fffffc804af2e3710
Flags	[Unrecognized flags 399b9020] AF_CLOSING
References	fffffc804
Close Requested?	0
Miniport	0 - [Unreadable NetAdapter]
Call Manager	fffffc804b90a4ac0 - [Invalid Open]
Call Manager Context	007a0078
Client	00060000 - [Unreadable Open]
Client Context	fffffc804af2e3888

CLIENT HANDLERS

Client Handler available)	Function pointer	Symbol (if
C1CreateVcHandler	ffffff80965fd5888	
mrvlpcie8897!Globals+8		
C1DeleteVcHandler	[None]	
C1RequestHandler	fffffc804af96fd78	
C1RequestCompleteHandler	fffffc804af96fd78	
C1OpenAfCompleteHandler	[None]	
C1CloseAfCompleteHandler	[None]	
C1RegisterSapCompleteHandler	000132060098028a	
C1DeregisterSapCompleteHandler	[None]	
C1MakeCallCompleteHandler	ffffff80965ff9ec0	

wdiwifi!MPWrapperSetOptions	
ClCloseCallCompleteHandler	fffff80965ff9c70
wdiwifi!MPWrapperHalt	
ClModifyCallQoSCompleteHandler	fffff80965ff9b50
wdiwifi!MPWrapperInitializeEx	
ClAddPartyCompleteHandler	fffff80965e71a08
mrvlpcie8897!MPUnload	
ClDropPartyCompleteHandler	fffff80965ffa070
wdiwifi!MPWrapperPause	
ClIncomingDropPartyHandler	fffff80965ffaa20
wdiwifi!MPWrapperReturnNetBufferLists	
ClIncomingCallHandler	fffff80965ffa1e0
wdiwifi!MPWrapperRestart	
ClCallConnectedHandler	fffff80965ffaad0
wdiwifi!MPWrapperCancelSendNetBufferLists	
ClIncomingCloseCallHandler	fffff80965ffa870
wdiwifi!MPWrapperSendNetBufferLists	
ClIncomingCallQoSChangeHandler	fffff80965ffa610
wdiwifi!MPWrapperOidRequest	

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Connection-Oriented NDIS](#)

[Address Families](#)

!ndiskd.ndisref

Article • 10/25/2023

The **!ndiskd.ndisref** extension displays a debug log of a tracked reference count.

Console

```
!ndiskd.ndisref -handle <x> [-tagtype <str>] [-stacks] [-tag <str>] [-refdebug]
```

Parameters

-handle

Required. Handle of the refcount block.

-tagtype

Enum type of the tags.

-stacks

Includes stack traces (if available).

-tag

Limits display to a single tag.

-refdebug

Shows detailed debug log, if available.

DLL

Ndiskd.dll

Examples

The following example passes the handle of an NDIS miniport driver to the **!ndiskd.ndisref** extension to display the refcount block for that driver. First, run [!ndiskd.minidriver](#) with no parameters to see a list of all miniport drivers on the system. In the example output below, look for the handle for the kdnic driver, fffffdf8015a98380.

Console

```
3: kd> !ndiskd.minidriver
fffffdf8015a98380 - tunnel
```

Using the miniport driver's handle, enter the !ndiskd.ndisref -handle command to see the refcount block for this miniport driver. The following example has the middle of the refcount block excised for brevity.

```
Console

3: kd> !ndiskd.ndisref fffffdf801418d650

REFCOUNT BLOCK

  Tag           Number of references
  0n0           0n6293          Show stacks
  0n1           0n1045          Show stacks
  0n2           0n4294966717    Show stacks
  0n5           0n25048         Show stacks
  0n18          0n4294967293    Show stacks
  0n19          0n4294967295    Show stacks
  0n21          0n4294967036    Show stacks
  0n23          0n30818         Show stacks
  0n24          0n24693         Show stacks
  0n25          0n25808         Show stacks

  ...
  0n153          0n7            Show stacks
  0n154          0n3            Show stacks
  0n156          0n29972        Show stacks
  0n159          0n4294959128    Show stacks
  0n160          0n30892        Show stacks
  0n161          0n136          Show stacks
  0n162          0n4294951910    Show stacks
  0n163          0n30892        Show stacks
  0n164          0n136          Show stacks
  0n167          0n4294965996    Show stacks

Include inactive tags
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[**!ndiskd.help**](#)

[**!ndiskd.minidriver**](#)

!ndiskd.ndisevent

Article • 10/25/2023

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisevent** extension displays an NDIS debug event log.

```
Console
!ndiskd.ndisevent -handle <x> [-tagtype <str>]
```

Parameters

-handle

Required. Handle of the event log.

-tagtype

Enum type of the tags.

DLL

Ndiskd.dll

Examples

To see the output of the event log for a network adapter, !ndiskd provides a link to it in the State section of the [!ndiskd.netadapter](#) output. This is easier than the manual method of finding an event log's handle from a miniport block and using that to run the **!ndiskd.ndisevent** extension.

First, enter the **!ndiskd.netadapter** command with no parameters to see a list of network adapters and miniport drivers on the system. In the following example, look for the handle for the Marvell AVASTAR Wireless-AC Network Controller, fffffc804b9e6f1a0.

```
Console
1: kd> !ndiskd.netadapter
      Driver          NetAdapter        Name
      fffffc804af2e3710  fffffc804b9e6f1a0  Marvell AVASTAR Wireless-AC
      Network Controller
```

fffffc804b99b9020	fffffc804b9c301a0	WAN Miniport (Network Monitor)
fffffc804b99b9020	fffffc804b9c2a1a0	WAN Miniport (IPv6)
fffffc804b99b9020	fffffc804b8a8a1a0	WAN Miniport (IP)
fffffc804ae9d7020	fffffc804b9ceb1a0	WAN Miniport (PPPOE)
fffffc804b9ca5900	fffffc804b9e601a0	WAN Miniport (PPTP)
fffffc804b99dc720	fffffc804b99b01a0	WAN Miniport (L2TP)
fffffc804b86581b0	fffffc804b9c6c1a0	WAN Miniport (IKEv2)
fffffc804ad4a7250	fffffc804b99651a0	WAN Miniport (SSTP)
fffffc804b11c4020	fffffc804b85821a0	Microsoft ISATAP Adapter
fffffc804b11c4020	fffffc804b71731a0	Microsoft ISATAP Adapter #2
fffffc804ad725020	fffffc804b05e71a0	Surface Ethernet Adapter #2
fffffc804b0bf0020	fffffc804b0c011a0	Bluetooth Device (Personal Area Network)
fffffc804aef695e0	fffffc804aed331a0	TAP-Windows Adapter V9

Now, click on the link for that NetAdapter or enter the `!ndiskd.netadapter -handle` command to see its details. Look for the "Show state history" link to the right of the Device PnP field, in the State section.

Console

```
1: kd> !ndiskd.netadapter fffffc804b9e6f1a0
```

MINIPORT

```
Marvell AVASTAR Wireless-AC Network Controller
```

Ndis handle	fffffc804b9e6f1a0
Ndis API version	v6.50
Adapter context	fffffc804af3b1100
Driver	fffffc804af2e3710 - mrvlpcie8897 v1.0
Network interface	fffffc804aea60a20
Media type	802.3
Physical medium	NdisPhysicalMediumUnspecified
Device instance	
PCI\VEN_11AB&DEV_2B38&SUBSYS_045E0001&REV_00\4&379f07b2&0&00E0	
Device object	fffffc804b9e6f050 More information
MAC address	c0-33-5e-13-22-f7

STATE

Miniport	INITIALIZING
Device PnP	ADDED Show state history
Datapath	Normal
Operational status	DOWN
Operational flags	[No flags set]
Admin status	ADMIN_UP
Media	MediaConnectUnknown
Power	D0

References	1	Show detail
Total resets	0	
Pending OID	None	
Flags	IN_INITIALIZE, NOT_BUS_MASTER,	
DEFAULT_PORT_ACTIVATED,		
	NOT_SUPPORTS_MEDIA_SENSE, DOES_LOOPBACK,	
MEDIA_CONNECTED		
PnP flags	PM_SUPPORTED, RECEIVED_START, HARDWARE_DEVICE	

WDI

This system supports WDI.
[Learn more about the associated WDI state](#)

BINDINGS

Protocol list	Driver	Open	Context
No protocols are bound to this miniport			
Filter list	Driver	Module	Context
No filters are bound to this miniport			

MORE INFORMATION

Driver handlers	Task offloads
Power management	PM protocol offloads
Pending OIDs	Timers
Pending NBLs	Receive side throttling
Wake-on-LAN (WoL)	Packet filter
Receive queues	Receive filtering
RSS	NIC switch
Hardware resources	Selective suspend
NDIS ports	WMI guides
Diagnostic log	

Now you can click the "Show state history" link or use the net adapter's handle to enter the **!ndiskd.netadapter -handle -log** command, which will show you the PnP event log for this miniport's miniport driver.

Console

```
1: kd> !ndiskd.netadapter fffffc804b9e6f1a0 -log
```

MINIPORT PM & PNP EVENTS

Event	Timestamp	(most recent event at bottom)
DeviceAdded		

13 ms later

DeviceStart

Mon Mar 20 21:27:07.106 2017 (UTC - 7:00) Now?

Set a breakpoint on the next event

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[!ndiskd.netadapter](#)

!ndiskd.ndisstack

Article • 10/25/2023

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisstack** extension displays a debug stack trace.

```
Console  
!ndiskd.ndisstack -handle <x> [-statistics]
```

Parameters

-handle

Required. Handle of the stack block.

-statistics

Shows debugging statistics.

DLL

Ndiskd.dll

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.wdiadapter

Article • 10/25/2023

The **!ndiskd.wdiadapter** extension displays information about a WDI WiFi!CAdapter structure. If you run this extension with no parameters, !ndiskd will display a list of all WDI WiFi!CAdapter structures.

For more information about WDI miniport drivers, see the [WDI Miniport Driver Design Guide](#).

For more information about WDI miniport driver reference, see [WDI Miniport Driver Reference](#).

Console

```
!ndiskd.wdiadapter [-handle <x>] [-pm] [-rcvfilter]
```

Parameters

-handle

Optional handle of a CAdapter object.

-pm

Shows power management state and capabilities.

-rcvfilter

Shows receive filtering capabilities.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.wdiadapter** extension with no parameters to see a list of all CAdapter objects, along with details for each of their WDI Adapters. In the following example, there is only one CAdapter structure. The handle for its associated WDI Adapter is fffffc804af396000.

Console

```
1: kd> !ndiskd.wdiadapter
      CAdapter
      fffffc804af396000 - WDI Adapter
```

WDI ADAPTER

Object	fffffc804af396000
Miniport	fffffc804b9e6f1a0
WDI driver	fffffc804b8ce7c40 - WDI MiniDriver
NetGuid	47cb3aa5-e29c-4628-80bd-5023c53fcccb
Power	D0
CCtlPlane	fffffc804af396080
CTxMgr	fffffc804af399790

DEVICE COMMANDS

```
Scheduler state    DeviceCommandSchedulerStateInit
[No queued device commands found]
```

Device Command Scheduler

SERIALIZED JOBS

Job	State	Type	Child
fffffc804b9f33000	StepPending	WiFiJobMiniportInitialize	

ACTIVE JOBS

Job	State	Type	Child
fffffc804b9f33000	StepPending	WiFiJobMiniportInitialize	

EVENTS

```
Processing?        No
```

Event	Status	Type
[No events found]		

Event Queue

MORE INFORMATION

Power management

Receive filtering

Now you can either click on the "Power management" and "Receive filtering" links at the bottom of the CAdapter structure's description, or you can enter the `!ndiskd.wdiadapter -handle` command with either the `-pm` or `-rcvfilter` option. The following example shows output for the `-rcvfilter` option.

```
Console

1: kd> !ndiskd.wdiadapter fffffc804af396000 -rcvfilter

RECEIVE FILTER

Current Configuration

Revision          0
Flags             0
Enabled filter types [No flags set]
Enabled queue types [No flags set]
Max Queues        0
Queue properties   [No flags set]
Filter tests      [No flags set]
Header types      [No flags set]
MAC header fields [No flags set]
Max MAC header filters 0
Max queue groups  0
Max queues per group 0
Min lookahead split size 0
Max lookahead split size 0
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[WDI Miniport Driver Design Guide](#)

[WDI Miniport Driver Reference](#)

!ndiskd.wdiminidriver

Article • 10/25/2023

The **!ndiskd.wdiminidriver** extension displays information about one or more CMiniportDriver structures. If you run this extension with no parameters, !ndiskd will display a list of all CMiniportDriver structures.

For more information about WDI miniport drivers, see the [WDI Miniport Driver Design Guide](#).

For more information about WDI miniport driver reference, see [WDI Miniport Driver Reference](#).

Console

```
!ndiskd.wdiminidriver [-handle <x>] [-pm] [-rcvfilter]
```

Parameters

-handle

Optional handle of a CMiniportDriver object.

-basic

Displays basic information about the miniport driver.

-handlers

Displays this driver's miniport handlers.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.wdiminidriver** extension with no parameters to see a list of all CMiniportDriver objects. In the following example, there is only one CMiniportDriver object. The handle of its WdiMiniDriver is fffffc804b8ce7c40.

Console

```
1: kd> !ndiskd.wdiminidriver
      WdiMiniDriver      Name
      fffffc804b8ce7c40 - WDI MiniDriver
```

Click on the WdiMiniDriver's handle or enter the **!ndiskd.wdiminidriver -handle** command to see details for this WDI miniport driver.

Console

```
1: kd> !ndiskd.wdiminidriver fffffc804b8ce7c40
```

WDI MINIPORT DRIVER

Object	fffffc804b8ce7c40
DRIVER_OBJECT	fffffc804b90a4ac0
Ndis API version	v6.50
NDIS driver	fffffc804af2e3710 - mrvlpcie8897
Driver version	v1.0
WDI API version	v1.0.1

Handlers

Now you can click the "Handlers" link at the bottom of the WDI Miniport Driver's details, or you can use the handle to enter the **!ndiskd.wdiminidriver -handle -handlers** command, to see a list of all this WDI Miniport Driver's handlers.

Console

```
1: kd> !ndiskd.wdiminidriver fffffc804b8ce7c40 -handlers
```

HANDLERS

Miniport Handler available)	Function pointer	Symbol (if available)
InitializeHandlerEx	[None]	
SetOptionsHandler	fffffc80965e8cae0	
mrvlpcie8897!wlan_cmd_queue_deinit		
UnloadHandler	fffffc80965e71a08	
mrvlpcie8897!MPUnload		
HaltHandlerEx	[None]	
ShutdownHandlerEx	fffffc80965e71974	
mrvlpcie8897!MPShutdown		
CheckForHangHandlerEx	[None]	
ResetHandlerEx	[None]	

PauseHandler	[None]
RestartHandler	[None]
OidRequestHandler	[None]
SendNetBufferListsHandler	[None]
ReturnNetBufferListsHandler	[None]
CancelSendHandler	[None]
CancelOidRequestHandler	[None]
DirectOidRequestHandler	[None]
CancelDirectOidRequestHandler	[None]
DevicePnPEventNotifyHandler	fffff80965e71868
mrvlpcie8897!MPPnPEventNotify	
AllocateAdapterHandler	fffff80965e713ec
mrvlpcie8897!MPAllocateTalAdapter	
FreeAdapterHandler	fffff80965e717e8
mrvlpcie8897!MPFreeTalAdapter	
OpenAdapterHandler	fffff80965e719b8
mrvlpcie8897!MPTaskOpenHandler	
CloseAdapterHandler	fffff80965e719ac
mrvlpcie8897!MPTaskCloseHandler	
StartOperationHandler	[None]
StopOperationHandler	[None]
PostPauseHandler	[None]
PostRestartHandler	[None]
HangDiagnoseHandler	fffff80965e715d8
mrvlpcie8897!MPDiagnosticHandler	
TalTxRxInitializeHandler	fffff80965e752d4
mrvlpcie8897!TalDataPathInitialize	
TalTxRxDeinitializeHandler	fffff80965e7527c
mrvlpcie8897!TalDataPathDeinitialize	
OpenAdapterCompleteHandler	fffff80965ffab50
wdiwifi!WDIOpenAdapterCompleteHandler	
CloseAdapterCompleteHandler	fffff80965fface0
wdiwifi!WDICloseAdapterCompleteHandler	

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[WDI Miniport Driver Design Guide](#)

WDI Miniport Driver Reference

!ndiskd.nwadapter

Article • 10/25/2023

The **!ndiskd.nwadapter** extension displays information about one or more nwifi!ADAPT structures. If you run this extension with no parameters, !ndiskd will display a list of all nwifi!ADAPT structures.

Console

```
!ndiskd.nwadapter [-handle <x>]
```

Parameters

-handle

Address of an ADAPT block.

DLL

Ndiskd.dll

See Also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

!ndiskd.ndisrwlock

Article • 10/25/2023

The **!ndiskd.ndisrwlock** extension displays information about an [NDIS_RW_LOCK_EX](#) lock structure.

Console

```
!ndiskd.ndisrwlock -handle <x>
```

Parameters

-handle

Required. Handle of the lock structure.

DLL

Ndiskd.dll

Examples

Use the **!ndiskd.ndisrwlock** extension if you create your own RW lock and would want to inspect it. To obtain the handle for an RW lock, use the *poi* command to dereference the address of your driver's lock. The following snippet shows how to look at a lock that the TCPIP protocol was using at the time of the example.

Console

```
0: kd> !ndiskd.ndisrwlock poi(tcpip!gALeHashtableLock)
```

NDIS READ-WRITE LOCK

Allocated by	[NDIS generic object]
Exclusive access	Not acquired
Read-only access	0 references

Set a breakpoint on acquire/release

To observe the driver using this RW lock, click on the "Set a breakpoint on acquire/release" link at the bottom of the RW lock's details. After setting the breakpoint, enter the **g** command to let the debuggee machine run and hit the breakpoint.

Console

```
0: kd> ba r4 fffffe00bc3fc22f8
0: kd> g
Breakpoint 0 hit
nt!KeTestSpinLock+0x3:
fffff802`0d69eb53 4885c0        test    rax,rax
```

Now you can re-run the same `!ndiskd.ndisrwlock` command to see that this RW lock has one Read-only access reference.

Console

```
0: kd> !ndiskd.ndisrwlock poi(tcpip!gAleHashtableLock)

NDIS READ-WRITE LOCK

Allocated by      [NDIS generic object]
Exclusive access Not acquired
Read-only access  1 reference

Set a breakpoint on acquire/release
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[NDIS_RW_LOCK_EX](#)

!ndiskd.ndisslot

Article • 10/25/2023

Note Third party network driver developers are not expected to manually use this extension command. You can run it to see the information it displays but you are not able to reuse the details it provides in your driver.

The **!ndiskd.ndisslot** extension displays the contents of an NDIS per-processor variable. If you run this extension with no parameters, !ndiskd will display a list of all NDIS per-processor variables on the system.

Console

```
!ndiskd.ndisslot [-handle <x>] [-itemtype <str>]
```

Parameters

-handle

Handle of the slot.

-itemtype

Type of the value stored in the slot.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.ndisslot** extension with no parameters to see a list of all per-processor slot variables. The following example output has excised the middle portion of the list for brevity.

Console

```
1: kd> !ndiskd.ndisslot
      Per-processor slot          Summary of contents
      fffffc804ae060000 - NDrw    All values are zero
      fffffc804ae060008 - NDrw    All values are zero
      fffffc804ae060010 - NDrw    All values are zero
      fffffc804ae060018 - NDrw    All values are zero
      fffffc804ae060020 - NDrw    All values are zero
```

fffffc804ae060028	- NDrw	All values are zero
fffffc804ae060030	- NDrw	All values are zero
fffffc804ae060038	- NDrw	All values are zero
fffffc804ae060040	- NDrw	All values are zero
fffffc804ae060048	- NDrw	All values are zero
...		
fffffc804ae060910	- NDtk	All values are zero
fffffc804ae060918	- NDtk	All values are zero
fffffc804ae060920	- tsR	All values are non-zero
fffffc804ae060928	- NDrw	All values are zero
fffffc804ae060930	- NDtk	All values are zero
fffffc804ae060938	- NDtk	All values are zero
fffffc804ae060940	- NDtk	All values are zero
fffffc804ae060948	- NDtk	All values are zero
fffffc804ae060950	- NDrw	All values are zero
fffffc804ae060958	- NDrw	All values are zero
Statistics		
Descriptors	1	
Total slots	512	
Slots available	275	
Slots used	237	
Efficiency	46%	

Clicking on one of the handles for the per-processor slot variables will show you the details for that variable. The following example uses the handle fffffc804ae060920 for the tsR variable, from the previous example.

Console

```
1: kd> !ndiskd.ndisslot fffffc804ae060920
Processor      Slot value
00            00000006
01            00000006
02            00000006
03            00000006
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

Indiskd.help

!ndiskd.ndis

Article • 10/25/2023

The **!ndiskd.ndis** extension displays build information about ndis.sys.

Console

```
!ndiskd.ndis
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

The following example shows that the debuggee machine has a Checked Ndis build.

Console

```
0: kd> !ndiskd.ndis
Checked Ndis
```

!ndiskd.dbglevel

Article • 10/25/2023

The **!ndiskd.dbglevel** extension displays and optionally changes the current NDIS debug level.

Warning **!ndiskd.dbglevel** has been superceded by WPP (Windows software trace preprocessor) and Driver Verifier. **!ndiskd** will give you the following warning if your target system does not support **!ndiskd.dbglevel**.

```
Console

0: kd> !ndiskd.dbglevel
This target does not support tracing through !ndiskd.dbglevel or
!ndiskd.dbgsystems.
Learn how to collect traces with WPP
```

If you click on the link at the bottom of the warning, **!ndiskd** will give you more information.

```
Console

0: kd> !ndiskd.help wpptracing
WPP traces are fast, flexible, and detailed. Plus, starting with
Windows 8
    and Windows Server 2012, you can automatically decode NDIS traces using
the
    symbol file. Just point TraceView (or tracepdb.exe) at NDIS.PDB, and it
will be able to get all the TMFs it needs to trace NDIS activity.

    If you would like traces to be printed in the debugger window, you use
the
    !wmitrace extension. For example, you might enable traces with this:

    !wmitrace.searchpath c:\path\to\TMF\files
    !wmitrace.start ndis -kd
    !wmitrace.enable ndis {DD7A21E6-A651-46D4-B7C2-66543067B869} -level 4 -
flag 0x31f3
```

For more information about WPP, see [WPP Software Tracing](#).

For more information about Driver Verifier, see [Driver Verifier](#).

For more information about WMI tracing, see [WMI Tracing Extensions \(Wmitrace.dll\)](#).

```
Console
```

```
!ndiskd.dbglevel [-level <str>]
```

Parameters

-level

The level of debugging verbosity. Possible values are:

- NONE - disables debug tracing
- FATAL - enables fatal errors to be printed
- ERROR - enables errors to be printed
- WARN - enables warnings to be printed
- INFO - enables informational messages to be printed
- VERBOSE - enables all debug traces to be printed

DLL

Ndiskd.dll

Remarks

This extension applies to checked NDIS.sys only. To check the build info of NDIS.sys, run the [!ndiskd.ndis](#) extension.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[!ndiskd.ndis](#)

[WPP Software Tracing](#)

[Driver Verifier](#)

[WMI Tracing Extensions \(Wmitrace.dll\)](#)

!ndiskd.dbgsystems

Article • 10/25/2023

The **!ndiskd.dbgsystems** extension displays and optionally changes the NDIS subsystems that have debug traces enabled.

Warning **!ndiskd.dbgsystems** has been superceded by WPP (Windows software trace preprocessor) and Driver Verifier. !ndiskd will give you the following warning if your target system does not support **!ndiskd.dbgsystems**.

```
Console

0: kd> !ndiskd.dbgsystems
This target does not support tracing through !ndiskd.dbglevel or
!ndiskd.dbgsystems.
Learn how to collect traces with WPP
```

If you click on the link at the bottom of the warning, !ndiskd will give you more information.

```
Console

0: kd> !ndiskd.help wpptracing
WPP traces are fast, flexible, and detailed. Plus, starting with
Windows 8
    and Windows Server 2012, you can automatically decode NDIS traces using
the
    symbol file. Just point TraceView (or tracepdb.exe) at NDIS.PDB, and it
will be able to get all the TMFs it needs to trace NDIS activity.

    If you would like traces to be printed in the debugger window, you use
the
    !wmitrace extension. For example, you might enable traces with this:

    !wmitrace.searchpath c:\path\to\TMF\files
    !wmitrace.start ndis -kd
    !wmitrace.enable ndis {DD7A21E6-A651-46D4-B7C2-66543067B869} -level 4 -
flag 0x31f3
```

For more information about WPP, see [WPP Software Tracing](#).

For more information about Driver Verifier, see [Driver Verifier](#).

For more information about WMI tracing, see [WMI Tracing Extensions \(Wmitrace.dll\)](#).

```
Console
```

```
!ndiskd.dbgsystems [-subsystem <any>]
```

Parameters

-subsystem

The subsystem to toggle.

If multiple components are selected, separate them with spaces. If a previously-selected component is repeated, its debug monitoring will be toggled off. The following values are possible:

Value	Meaning
INIT	Traces adapter initialization.
CONFIG	Traces adapter configuration.
SEND	Traces sending data over the network.
RECV	Traces receiving data from the network.
PROTOCOL	Traces protocol operations.
BIND	Traces binding operations.
BUS_QUERY	Traces bus queries.
REGISTRY	Traces registry operations.
MEMORY	Traces memory management.
FILTER	Traces filter operations.
REQUEST	Traces requests.
WORK_ITEM	Traces work-item operations.
PNP	Traces Plug and Play operations.
PM	Traces Power Management operations.
OPEN	Traces operations that open reference objects.
LOCKS	Traces locking operations.
RESET	Traces resetting operations.
WMI	Traces Windows Management Instrumentation operations.

NDIS_CO	Traces Connection-Oriented NDIS.
REFERENCE	Traces reference operations.

DLL

Ndiskd.dll

Remarks

This extension applies to checked NDIS.sys only. To check the build info of NDIS.sys, run the [!ndiskd.ndis](#) extension.

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[!ndiskd.ndis](#)

[WPP Software Tracing](#)

[Driver Verifier](#)

[WMI Tracing Extensions \(Wmitrace.dll\)](#)

!ndiskd.ndiskdversion

Article • 10/25/2023

The **!ndiskd.ndiskdversion** extension displays information about !ndiskd itself.

```
Console  
!ndiskd.ndiskdversion
```

Parameters

This extension has no parameters.

DLL

Ndiskd.dll

Examples

The following example shows output for **!ndiskd.ndiskdversion**.

```
Console  
  
0: kd> !ndiskd.ndiskdversion  
NDISKD Version      17.08.00 (NDISKD codename "We'll deploy IPv6 real  
soon now")  
Build              Release  
Debugger CPU       AMD64  
Hyperlinks (DML)   Enabled  
Unicode            Enabled  
Debug NDISKD       Disabled
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

Indiskd.help

!ndiskd.netreport

Article • 10/25/2023

The **!ndiskd.netreport** extension generates a visual report of the entire network stack. The report **!ndiskd.netreport** generates is an HTML file, and it will give you a link to its location. The HTML file contains detailed information about the network stack, so if you need to share it for analysis you can email it instead of having to send a large crash dump file.

Console

```
!ndiskd.netreport [-outputpath <str>] [-jsononly]
```

Parameters

-outputpath

Specifies where to write the report file.

-jsononly

Only writes the raw data, no HTML.

DLL

Ndiskd.dll

Examples

Run the **!ndiskd.netreport** extension to draw a box diagram of your network stack.

Console

```
1: kd> !ndiskd.netreport
```

NETWORK STACK REPORT

Want more stuff? Rerun with the -verbose flag

Report was saved to C:\Users*****\AppData\Local\Temp\NKDFE9F.html
[View the report](#) [Send in email](#)

Click the "View the report" link at the bottom to see the report generated. The following image shows a net report generated from a crash dump file. Each vertical stack is a network adapter, broken down into layers showing the components of the stack. The color of each box is generated by hashing the name of the component, which means the same components will render with the same color every time you run the report. This means you can easily pick out a particular driver or adapter if you are debugging an issue with it.



As a comparison, the following image shows a net report generated from an active system instead of a crash dump file. Note that there are two more options at the bottom of the HTML page to "Show data flows" and "Simulate packets," and there is a fourth tab at the top of the report for "Data flows." These options appeared because the debuggee machine had NBL tracking enabled, which lets **!ndiskd.netreport** parse the NBL tracking log to display the information visually. If NBL tracking is not turned on, these options will not appear. For more information about NBL tracking and the NBL log, see [!ndiskd.nbllog](#).

By checking the "Show data flows" box, you can see the paths where the data is flowing. By checking the "Simulate packets" box, you can see animated circles moving up and down the data flow paths. Each circle represents a network packet.

NETWORK DEBUG REPORT

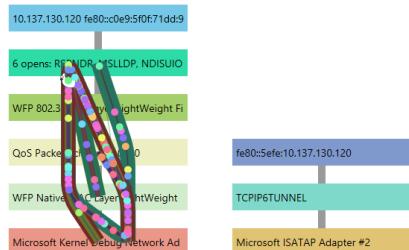
OVERVIEW

SYSTEM

SUMMARIES

DATA FLOWS

[Help](#)



Show hidden miniports Show NDIS filters Show IP addresses Show data flows Simulate packets

This second example from an active system also shows another difference from the first example, which used a crash dump file. The target debuggee machine in the second example was provisioned for kernel debugging over a network, so you can see the network adapter on the stack with the data flows is the Microsoft Kernel Debug Network Adapter. This adapter is usually hidden unless kernel debugging has been enabled on the debuggee machine. In reality, the Kernel Debug Network Adapter has reserved the machine's Ethernet adapter for the debug session, so traffic is flowing over Ethernet.

The ability to visualize the network stack and see where traffic is flowing can enable you to quickly identify where a problem might be. This can be particularly helpful for virtual switches or servers, which have more complicated network diagrams than the previous examples. For example, on a Windows Server that uses NIC Teaming, you can see if multiple network stacks cross each other to balance the traffic load and identify if there is an issue at the bottom of one stack that is affecting another stack. To see an example of a network debug report that shows this, see [Debugging the Network Stack](#). For more information about NIC Teaming, see [Using NIC Teaming for Network Subsystem Performance](#).

!ndiskd.netreport also has other tabs at the top of the page for System, Summaries, and Data Flows (if applicable). These tabs contain further useful information about the state of the network stack. The following image shows the Network Interfaces tab, under the Summaries tab. The table in this tab lets you see more information about the names and identifiers for the network interfaces in the system.

NETWORK DEBUG REPORT						
OVERVIEW		SYSTEM	SUMMARIES		DATA FLOWS	
NETWORK INTERFACES		MAC ADDRESSES		IPv4 & IPv6		
NetLuid	IfIndex	Guid			Name	H...
24:00	1	aa87e4cd-7b53-11e6-b714-806e6f6e6963			Software Loopback Interface 1	ffffe00...
06:8004	2	1fbfd86-3a57-4162-aa1e-125281a854c7			WAN Miniport (Network Monitor)	ffffe00...
23:8000	3	4090e55d-5787-47ea-af0b-41a355035edf			WAN Miniport (PPPOE)	ffffe00...
131:8002	4	461fc1e5-d6bf-43b4-9ae9-dfb7b6c4508			WAN Miniport (IKEv2)	ffffe00...
06:8002	5	48312685-138a-4170-9369-e8439c3a1eeb			WAN Miniport (IP)	ffffe00...
06:8000	6	551a82bc-0677-4fb9-a74b-f9cdba5f2b1e			Microsoft Kernel Debug Network Adapter	ffffe00...
131:8000	7	6730c246-c98a-427e-9976-8b29dbc6f18a			Microsoft ISATAP Adapter	ffffe00...
131:8001	8	6b204269-1a22-4fec-b672-cfa37a80f88d			WAN Miniport (SSTP)	ffffe00...
131:8005	9	6d567b1a-2189-4288-b4ce-6ec580e8a570			Microsoft ISATAP Adapter #2	ffffe00...
131:8004	10	755b22eb-a626-4575-b4f8-d2a4bc3eebb6			WAN Miniport (PPTP)	ffffe00...
06:8003	11	7e573422-226f-4cb9-b43c-9e0934af8c5			WAN Miniport (IPv6)	ffffe00...
131:8003	12	9bf86317-5c62-4fed-b177-50f723ab0f62			WAN Miniport (L2TP)	ffffe00...
131:8006	13	bcd40482-26ac-41dd-b5c9-5cf8a1510fe			Teredo Tunneling Pseudo-Interface	ffffe00...
06:8001	14	cbddfde1-5570-4c65-9d47-52d63abce00c			Intel(R) 82579LM Gigabit Network Connection	ffffe00...
06:00	15	fc2a0ae0-b103-11e6-b724-806e6f6e6963			Microsoft Kernel Debug Network Adapter-WFP Native MAC Layer LightWeight Filter-0000	ffffe00...
06:01	16	fc2a0ae1-b103-11e6-b724-806e6f6e6963			Microsoft Kernel Debug Network Adapter-QoS Packet Scheduler-0000	ffffe00...
06:02	17	fc2a0ae0-b103-11e6-b724-806e6f6e6963			Microsoft Kernel Debug Network Adapter-WFP 802.3 MAC Layer LightWeight Filter-0000	ffffe00...

+ Note that the IfIndex changes each time the network interface is started (or the computer is rebooted). Avoid using the IfIndex as a permanent identifier.

The Data flows tab, which appears if NBL tracking was enabled on the target system, shows a table of traffic events and details about each one. The following image shows the Data flows tab from the active system in the second example debug report described previously.

NETWORK DEBUG REPORT						
OVERVIEW		SYSTEM	SUMMARIES		DATA FLOWS	
Occurrences	Weighted	Failure?	Event	Context	Source	Destination
855	30.7%		Freed			
327	11.8%		MiniportReceived	ffffe00bc53ae181	ffffe00bc5ac5c70	
			FilterReceived	00000020		
			IndicatedToProtocol_Synchronous	ffffe00bc30abb1	ffffe00bc30abc10	
			ProtocolReturned_Synchronous	ffffe00bc53ae181	ffffe00bc5ac5c70	
			IndicatedToProtocol	ffffe00bc5813971	ffffe00bc5ac0010	
			ProtocolReturned	ffffe00bc53ae181	ffffe00bc5ac5c70	
			FilterReturned	ffffe00bc51a7971	ffffe00bc3f701a0	
314	11.3%		Allocated			
			ProtocolSent	ffffe00bc4af2861	ffffe00bc5ab7c10	ffffe00bc5ac4880
			FilterSent	ffffe00bc53ae181	ffffe00bc5ac5c70	
			FilterSent	ffffe00bc51a7971	ffffe00bc3f701a0	
			SentToMiniport	ffffe00bc51a7971	ffffe00bc3f701a0	
			MiniportSendCompleted	ffffe00bc53ae181	ffffe00bc5ac5c70	
			FilterSendCompleted	ffffe00bc4af2861	ffffe00bc5ac4880	
			FilterSendCompleted	00000030		
			SendCompleted	ffffe00bc5ab88a1	ffffe00bc5ab7c10	
297	10.7%		MiniportReceived	ffffe00bc53ae181	ffffe00bc3f701a0	ffffe00bc5ac5c70
			FilterReceived	00000020		
			IndicatedToProtocol_Synchronous	ffffe00bc30abb1	ffffe00bc30abc10	
			ProtocolReturned_Synchronous	ffffe00bc53ae181	ffffe00bc5ac5c70	
			IndicatedToProtocol	ffffe00bc5813971	ffffe00bc5ac0010	
			ProtocolReturned	ffffe00bc53ae181	ffffe00bc5ac5c70	
			FilterReturned	ffffe00bc51a7971	ffffe00bc3f701a0	

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[!ndiskd.nbllog](#)

[Using NIC Teaming for Network Subsystem Performance](#)

!ndiskd.cxadapter

Article • 10/25/2023

The **!ndiskd.cxadapter** extension displays information about a NETADAPTER object.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#).

Console

```
!ndiskd.cxadapter [-handle <x>] [-basic] [-power] [-datapath]
```

Parameters

-handle

Required. Handle of a NETADAPTER.

-basic

Displays basic information.

-power

Displays information about the NETPOWERSETTINGS object of the NETADAPTER.

-datapath

Displays information about the datapath queues.

DLL

Ndiskd.dll

Examples

To obtain a handle for a NETADAPTER object, first run the [!ndiskd.netadapter](#) command to see a list of all NIC drivers and NetAdapters on the system. In the following example, look for the handle for the NetAdapter named Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2. Its handle is fffffd1022d048030.

Console

```
0: kd> !ndiskd.netadapter
Driver          NetAdapter        Name

```

fffffd1022e8ecae0	fffffd1022d048030	Realtek PCIe GBE Family
Controller NetAdapter	Sample Driver #2	
fffffd1022ed908e0	fffffd1022e8611a0	Microsoft Kernel Debug Network
Adapter		

By clicking on this NetAdapter's handle or by entering the !ndiskd.netadapter -handle command with its handle on the command line, you can see details for this NetAdapter, including its NETADAPTER object. The Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2's NETADAPTER handle is 00002efdd0e5f988.

Console

```
0: kd> !ndiskd.netadapter fffffd1022d048030
```

NETADAPTER

```
Realtek PCIe GBE Family Controller NetAdapter Sample Driver #2
```

Ndis handle	fffffd1022d048030	
NETADAPTER	00002efdd0e5f988	More information
WDFDEVICE	00002efdcf45f2f8	
Driver	fffffd1022e8ecae0	- RtEthSample v0.0
Network interface	fffffd1022e395a20	

Media type	802.3
------------	-------

Device instance	
-----------------	--

PCI\VEN_10EC&DEV_8168&SUBSYS_816810EC&REV_03\4&22bb23f1&0&0038	
--	--

Device object	fffffd1022de127f0	More information
---------------	-------------------	------------------

MAC address	00-e0-4c-68-00-8b
-------------	-------------------

STATE

Miniport	Running	
Device PnP	Started	Show state history
Datapath	Normal	
Interface	Up	
Media	Connected	
Power	D0	
References	0n10	Show detail
Total resets	0	
Pending OID	None	
Flags	NOT_BUS_MASTER, WDF, DEFAULT_PORT_ACTIVATED, SUPPORTS_MEDIA_SENSE, DOES_NOT_DO_LOOPBACK, MEDIA_CONNECTED	
PnP flags	PM_SUPPORTED, DEVICE_POWER_ENABLED, DEVICE_POWER_WAKE_ENABLE, HARDWARE_DEVICE, NDIS_WDM_DRIVER, WAKE_CAPABLE	

IP ADDRESS SUMMARY

10.137.188.169
adapter
fe80::3cad:81bb:5dad:1066

See all IP addresses on this

BINDINGS

Protocol list	Driver	Open	Context
MSLLDP	fffffd1023043f6a0	fffffd1022e786a90	
fffffd102307465c0			
LLTDIO	fffffd1022c6b7830	fffffd1022ef8cc00	
fffffd1022f1e5730			
TCPIP6	fffffd1022e2c7c10	fffffd10230b98310	
fffffd102304d9010			
(RASPPPOE)	Not running		
(RDMANDK)	fffffd1022d574a70	Declined with	
NDIS_STATUS_NOT_RECOGNIZED			
RSPNDR	fffffd1022c71a830	fffffd1022de0cc00	
fffffd1022d03f6a0			
TCPIP	fffffd1022e2cbc10	fffffd1022de067f0	
fffffd1022d03f010			
NDISUIO	fffffd1022de07670	fffffd1022cd648d0	
fffffd10231131970			

Filter list	Driver	Module	Context
WFP 802.3 MAC Layer LightWeight Filter-0000			
fffffd10230cf02c0	fffffd1022e384d70	fffffd1022f271660	
QoS Packet Scheduler-0000			
fffffd10231778700	fffffd1022d56f220	fffffd1022f26d660	
WFP Native MAC Layer LightWeight Filter-0000			
fffffd1022ed59c20	fffffd1022e384ad0	fffffd1022f26b660	

MORE INFORMATION

Driver handlers	Task offloads
Power management	PM protocol offloads
Pending OIDs	
Pending NBLs	Receive side throttling
Wake-on-LAN (WoL)	Packet filter
Receive queues	Receive filtering
RSS	NIC switch
	Selective suspend
NDIS ports	WMI guides
Diagnostic log	

Because the NETADAPTER object is a WDF object, clicking its handle will cause the debugger to run the [!wdfkd.wdfhandle](#) command which will give you more information

about it from a WDF perspective. To see more detailed information about the NETADAPTER from a networking perspective, click the "More Information" link to the right of the NETADAPTER's handle to run the **!ndiskd.cxadapter** command with its handle.

```
Console

0: kd> !ndiskd.cxadapter fffffd1022f1a0720

NETADAPTER

    Miniport      fffffd1022d048030 - Realtek PCIe GBE Family Controller
NetAdapter Sample Driver #2
    NETADAPTER    00002efdd0e5f988
    WDFDEVICE     00002efdcf45f2f8

    Event Callbacks           Function pointer   Symbol (if
available)
        EvtAdapterCreateTxQueue    fffff80034151508
RtEthSample+1508
        EvtAdapterCreateRxQueue    fffff800341510ec
RtEthSample+10ec

    Show datapath info
    Show NETPOWERSETTINGS info
```

You can also combine this command other parameters such as **-datapath** to see more information for this NETADAPTER.

```
Console

0: kd> !ndiskd.cxadapter fffffd1022f1a0720 -basic -datapath

NETADAPTER

    Miniport      fffffd1022d048030 - Realtek PCIe GBE Family Controller
NetAdapter Sample Driver #2
    NETADAPTER    00002efdd0e5f988
    WDFDEVICE     00002efdcf45f2f8

    Event Callbacks           Function pointer   Symbol (if
available)
        EvtAdapterCreateTxQueue    fffff80034151508
RtEthSample+1508
        EvtAdapterCreateRxQueue    fffff800341510ec
RtEthSample+10ec

DATAPATH QUEUES
```

NETTXQUEUE
NETRXQUEUE

fffffd1022f512700
fffffd1022cc7b0d0

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

[!ndiskd.netadapter](#)

[!wdfkd.wdfhandle](#)

!ndiskd.netqueue

Article • 10/25/2023

The **!ndiskd.netqueue** extension displays information about a NETTXQUEUE or NETRXQUEUE object.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#).

```
Console  
!ndiskd.netqueue -handle <x> [-basic]
```

Parameters

-handle

Required. Handle of a NETTXQUEUE or NETRXQUEUE.

-basic

Displays basic information.

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NETTXQUEUE and NETRXQUEUE objects with other objects in the NetAdapterCx.

To obtain a handle for a NETTXQUEUE or NETRXQUEUE, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.
4. Enter the [!ndiskd.cxadapter](#) command with the *-datapath* parameter to see that NETADAPTER's datapath queues.

For details on this procedure, see the examples on the [!ndiskd.cxadapter](#) topic. In the following example, look for the handle for this NETADAPTER's NETTXQUEUE,

fffffd1022f512700.

Console

```
0: kd> !ndiskd.cxadapter fffffd1022f1a0720 -basic -datapath
```

NETADAPTER

Miniport	fffffd1022d048030	- Realtek PCIe GBE Family Controller
NetAdapter Sample Driver #2		
NETADAPTER	00002efdd0e5f988	
WDFDEVICE	00002efdcf45f2f8	

Event Callbacks available)	Function pointer	Symbol (if
EvtAdapterCreateTxQueue RtEthSample+1508	fffff80034151508	
EvtAdapterCreateRxQueue RtEthSample+10ec	fffff800341510ec	

DATAPATH QUEUES

NETTXQUEUE	fffffd1022f512700
NETRXQUEUE	fffffd1022cc7b0d0

By clicking on the NETTXQUEUE's handle or entering the **!ndiskd.netqueue -handle** command on the command line, you can see details for this queue, including the handle to its companion WDF object, the handle to its ring buffer, and function pointers for its registered callbacks.

Console

```
0: kd> !ndiskd.netqueue fffffd1022f512700
```

NETTXQUEUE	00002efdd0aed9a8
Ring buffer	fffffd1022d000000

Switch to EC thread

Event Callbacks available)	Function pointer	Symbol (if
EvtQueueAdvance RtEthSample+2af8	fffff80034152af8	
EvtQueueArmNotification RtEthSample+9a94	fffff80034159a94	
EvtQueueCancel RtEthSample+98d8	fffff800341598d8	

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[**!ndiskd.help**](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

[Summary of Objects](#)

[**!ndiskd.netadapter**](#)

[**!ndiskd.cxadapter**](#)

!ndiskd.netrb

Article • 10/25/2023

The **!ndiskd.netrb** extension displays information about a [NET_RING_BUFFER](#) structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#).

Console

```
!ndiskd.netrb -handle <x> [-basic] [-dump] [-elementtype <str>]
```

Parameters

-handle

Required. Address of a NET_RING_BUFFER.

-basic

Displays basic information.

-dump

Displays information about each element in the NET_RING_BUFFER.

-elementtype

A string for the data type to use when referring to a ring buffer element.

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NET_RING_BUFFER object with other objects in the NetAdapterCx.

To obtain a handle for a NET_RING_BUFFER, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.

4. Enter the **!ndiskd.cxadAPTER** command with the **-datapath** parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.

For details on Steps 1-4 of this procedure, see the examples on the **!ndiskd.cxadAPTER** topic. For details on Step 5 of this procedure, see the examples on the **!ndiskd.netqueue** topic. In the following example, look for the handle for this NETTXQUEUE's ring buffer, fffffd1022d000000.

```
Console

0: kd> !ndiskd.netqueue fffffd1022f512700

NETTXQUEUE          00002efdd0aed9a8
Ring buffer         fffffd1022d000000

Switch to EC thread

Event Callbacks           Function pointer   Symbol (if
available)
EvtQueueAdvance          ffffff80034152af8
RtEthSample+2af8
EvtQueueArmNotification  ffffff80034159a94
RtEthSample+9a94
EvtQueueCancel            ffffff800341598d8
RtEthSample+98d8
```

By clicking on the handle for the ring buffer or by entering the **!ndiskd.netrb -handle** command on the command line, you can see details for this NET_RING_BUFFER, including how many elements it contains and the address of its Begin and End indices.

```
Console

0: kd> !ndiskd.netrb fffffd1022d000000

NET_RING      fffffd1022d000000

Number of elements 0x080
Owned by OS       0x080
Owned by Client   00000

Begin Index      0x078 (fffffd1022d003c40 - NET_PACKET)
Next Index       0x078 (fffffd1022d003c40 - NET_PACKET)
End Index        0x078 (fffffd1022d003c40 - NET_PACKET)

List all elements
```

To see this NET_RING_BUFFER's elements, either click the "List all elements" link at the bottom of its details or enter the **!ndiskd.netrb -dump** command on the command line. The following example has had the middle elements excised for brevity.

```
Console

0: kd> !ndiskd.netrb fffffd1022d000000 -dump

[000] fffffd1022d000040 - NET_PACKET
[001] fffffd1022d0000c0 - NET_PACKET
[002] fffffd1022d000140 - NET_PACKET
[003] fffffd1022d0001c0 - NET_PACKET
[004] fffffd1022d000240 - NET_PACKET
[005] fffffd1022d0002c0 - NET_PACKET

...
[07b] fffffd1022d003dc0 - NET_PACKET
[07c] fffffd1022d003e40 - NET_PACKET
[07d] fffffd1022d003ec0 - NET_PACKET
[07e] fffffd1022d003f40 - NET_PACKET
[07f] fffffd1022d003fc0 - NET_PACKET
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

[Summary of Objects](#)

[NET_RING_BUFFER](#)

[!ndiskd.netadapter](#)

[!ndiskd.cxadapter](#)

[!ndiskd.netqueue](#)

!ndiskd.netpacket

Article • 10/25/2023

The **!ndiskd.netpacket** extension displays information about a [NET_PACKET](#) structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#).

Console

```
!ndiskd.netpacket -handle <x> [-basic] [-layout] [-checksum] [-data]
```

Parameters

-handle

Required. Address of a NET_PACKET.

-basic

Displays basic information.

-layout

Displays packet protocol layout.

-checksum

Displays packet checksum information.

-data

Dumps the payload memory.

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.

3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.
4. Enter the [!ndiskd.cxadapter](#) command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.
6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.

For details on Steps 1-4 of this procedure, see the examples on the [!ndiskd.cxadapter](#) topic. For details on Step 5 of this procedure, see the examples on the [!ndiskd.netqueue](#) topic. For details on Steps 6-7 of this procedure, see the examples on the [!ndiskd.netrb](#) topic. In the following example, look for the handle for the first NET_PACKET, fffffd1022d000040.

```
Console

0: kd> !ndiskd.netrb fffffd1022d000000 -dump

[000] fffffd1022d000040 - NET_PACKET
[001] fffffd1022d0000c0 - NET_PACKET
[002] fffffd1022d000140 - NET_PACKET
[003] fffffd1022d0001c0 - NET_PACKET
[004] fffffd1022d000240 - NET_PACKET
[005] fffffd1022d0002c0 - NET_PACKET

...
[07b] fffffd1022d003dc0 - NET_PACKET
[07c] fffffd1022d003e40 - NET_PACKET
[07d] fffffd1022d003ec0 - NET_PACKET
[07e] fffffd1022d003f40 - NET_PACKET
[07f] fffffd1022d003fc0 - NET_PACKET
```

By clicking on the handle for this NET_PACKET or by entering [!ndiskd.netpacket -handle](#) on the command line, you can see details for this NET_PACKET, including the ring buffer that contains it, the datapath queue that contains its ring buffer, and the handle for its first fragment.

```
Console

0: kd> !ndiskd.netpacket fffffd1022d000040

          NET_PACKET      fffffd1022d000040      Ring Buffer
fffffd1022d000000
First fragment      fffffd1022d000040      NETTXQUEUE
```

```
fffffd1022f512700
```

```
Client Context      fffffd1022d000090  
  
Show protocol layout  
Show checksum information  
Dump data payload
```

You can now combine the basic description with any of the other !ndiskd.netpacket parameters, or all of them, to see specific information for this fragment. The following example uses all parameters.

Console

```
0: kd> !ndiskd.netpacket fffffd1022d000040 -basic -layout -checksum -data  
  
NET_PACKET          fffffd1022d000040      Ring Buffer  
fffffd1022d000000  
First fragment      fffffd1022d000040      NETTXQUEUE  
fffffd1022f512700  
  
Client Context      fffffd1022d000090
```

Protocol Layout

```
Layer 2 Type        ETHERNET  
Header Length      0n14  
  
Layer 3 Type        IPV4_NO_OPTIONS  
Header Length      0n20  
  
Layer 4 Type        UDP  
Header Length       8
```

Checksum Information

```
Layer 2             TX_PASSTHROUGH  
Layer 3             TX_REQUIRED  
Layer 4             TX_PASSTHROUGH
```

Payload data

```
Fragment           fffffd1022d000040  
fffffd102303e8332  00 00 01 02 71 68 0a 89-be 39 e0 00 00 16 94 04  
....qh....9.....  
fffffd102303e8342  00 00 22 00 fa 01 00 00-00 01 03 00 00 00 e0 00  
..".....  
fffffd102303e8352  00 fc
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[**!ndiskd.help**](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

[Summary of Objects](#)

[NET_PACKET](#)

[**!ndiskd.netadapter**](#)

[**!ndiskd.cxadapter**](#)

[**!ndiskd.netqueue**](#)

[**!ndiskd.netrb**](#)

!ndiskd.netfragment

Article • 10/25/2023

The **!ndiskd.netfragment** extension displays information about a [NET_PACKET_FRAGMENT](#) structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#).

```
Console  
!ndiskd.netfragment -handle <xx>
```

Parameters

-handle

Required. Address of a NET_PACKET_FRAGMENT.

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.
4. Enter the [!ndiskd.cxadapter](#) command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.
6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.
8. Click on one of the [NET_PACKET](#) objects in the ring buffer's list of elements.

For details on Steps 1-4 of this procedure, see the examples on the [!ndiskd.cxadapter](#) topic. For details on Step 5 of this procedure, see the examples on the [!ndiskd.netqueue](#) topic. For details on Steps 6-7 of this procedure, see the examples on the [!ndiskd.netrb](#) topic. For details on Step 8 of this procedure, see the examples on the [!ndiskd.netpacket](#) topic. In the following example, look for the handle for the first fragment of this NET_PACKET, fffffd1022d000040.

```
Console

0: kd> !ndiskd.netpacket fffffd1022d000040

    NET_PACKET      fffffd1022d000040      Ring Buffer
fffffd1022d000000
    First fragment   fffffd1022d000040      NETTXQUEUE
fffffd1022f512700

    Client Context   fffffd1022d000090

    Show protocol layout
    Show checksum information
    Dump data payload
```

By clicking on the handle for the first fragment or by entering the [!ndiskd.netfragment -handle](#) command on the command line, you can see details for this NET_PACKET_FRAGMENT, including its Virtual Address, capacity, and whether or not it is the last packet in the NET_PACKET chain of fragments.

```
Console

0: kd> !ndiskd.netfragment fffffd1022d000040

    NET_PACKET_FRAGMENT fffffd1022d000040

    Virtual Address     fffffd102303e82f8
    Capacity            0n92
    Valid Length        0n34
    Offset              0n58

    Last packet of chain
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

Debugging the Network Stack

NDIS extensions (Ndiskd.dll)

!ndiskd.help

Network Adapter WDF Class Extension (Cx)

Summary of Objects

NET_PACKET_FRAGMENT

NET_PACKET

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netrb

!ndiskd.netpacket

!ndiskd.nrc

Article • 10/25/2023

The **!ndiskd.nrc** extension displays information about a [NET_RING_COLLECTION](#) structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#) and [Introduction to net rings](#).

```
Console  
!ndiskd.nrc -handle <xx> [-basic] [-packet] [-fragment] [-dump]
```

Parameters

-handle

Required. Address of NET_RING_COLLECTION

-basic

Displays links for packet ring and fragment ring.

-packet

Displays only the packet ring contents.

-fragment

Displays only the fragment ring contents.

-dump

Displays information about each element(packet/fragment).

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.

2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.
4. Enter the [!ndiskd.cxadapter](#) command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.
6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.
8. Click on one of the NET RING COLLECTION objects.

For details on Steps 1-4 of this procedure, see the examples on the [!ndiskd.cxadapter](#) topic. For details on Step 5 of this procedure, see the examples on the [!ndiskd.netqueue](#) topic. For details on Steps 6-7 of this procedure, see the examples on the [!ndiskd.netrb](#) topic.

In the following example, the handle for the NET_RING_COLLECTION, ffff8b82fbcf94b0 is used with the nrc command.

```
dbgcmd

0: kd> !ndiskd.nrc fffff8b82fbcf94b0

NET RING COLLECTION fffff8b82fbcf94b0

Packet Ring      fffff8b82f8e75000
Fragment Ring    fffff8b8300961000

List only packets
List only fragments
List all elements
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

Summary of Objects

NET_RING_COLLECTION

!ndiskd.netadapter

!ndiskd.cxadapter

!ndiskd.netqueue

!ndiskd.netrb

!ndiskd.netpacket

!ndiskd.netring

Article • 10/25/2023

The **!ndiskd.netring** extension displays information about a [NET_RING](#) structure.

For more information about the Network Adapter WDF Class Extension (NetAdapterCx), see [Network Adapter WDF Class Extension \(Cx\)](#) and [Introduction to net rings](#).

Console

```
!ndiskd.netring -handle <x> [-basic] [-dump]
```

Parameters

-handle

Required. Address of a NET_RING

-basic

Displays basic information

-dump

Displays information about each element

DLL

Ndiskd.dll

Examples

Note See [Summary of Objects](#) to see a diagram explaining the relationship of the NET_PACKET object with other objects in the NetAdapterCx.

To obtain a handle for a NET_PACKET, follow these steps:

1. Run the [!ndiskd.netadapter](#) extension.
2. Click on the handle for a NetAdapter that has a NetAdapterCx driver installed.
3. Click the "More Information" link to the right of the NetAdapter's NETADAPTER object to run the [!ndiskd.cxadapter](#) extension.
4. Enter the [!ndiskd.cxadapter](#) command with the *-datapath* parameter to see that NETADAPTER's datapath queues.
5. Click on the handle for one of the datapath queues.

6. Click on the handle for that datapath queue's ring buffer.
7. Click on the "List all elements" link at the bottom of the ring buffer details to see the elements it contains.
8. Click on one of the Net Ring Collection objects.

For details on Steps 1-4 of this procedure, see the examples on the [!ndiskd.cxadapter](#) topic. For details on Step 5 of this procedure, see the examples on the [!ndiskd.netqueue](#) topic. For details on Steps 6-7 of this procedure, see the examples on the [!ndiskd.netrb](#) topic.

This example shows the use of netring.

```
Console

0: kd> !ndiskd.netring fffff8b82f8e75000

NET RING          fffff8b82f8e75000

Begin index      000
Next index       000
End index        0x2
Reserved by OS   000
Element stride    0x40
Element index mask 0x7f

Number of elements 0x80
Owned by OS       0x7e
Owned by Client   0x2
```

See also

[Network Driver Design Guide](#)

[Windows Vista and Later Networking Reference](#)

[Debugging the Network Stack](#)

[NDIS extensions \(Ndiskd.dll\)](#)

[!ndiskd.help](#)

[Network Adapter WDF Class Extension \(Cx\)](#)

[Summary of Objects](#)

[NET_RING](#)

[**!ndiskd.netadapter**](#)

[**!ndiskd.cxadapter**](#)

[**!ndiskd.netqueue**](#)

[**!ndiskd.netrb**](#)

[**!ndiskd.netpacket**](#)

RPC Extensions (Rpcexts.dll)

Article • 10/25/2023

Extension commands that are useful for debugging Microsoft Remote Procedure Call (RPC) can be found in Rpcexts.dll.

The Windows XP and later version of this extension DLL appear in the winxp directory.

For more information about how to use these extensions, see [Using the RPC Debugger Extensions](#).

!rpcexts.eeinfo

Article • 04/03/2024

The !rpcexts.eeinfo extension displays the extended error information chain.

```
dbgcmd
```

```
!rpcexts.eeinfo EEInfoAddress
```

Parameters

EEInfoAddress

Specifies the address of the extended error information.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension displays the contents of all records in the extended error information chain.

The records are displayed in order, with the most recent records first. The records are separated by a line of dashes.

Here is an example (in which there is only one record):

```
dbgcmd
```

```
0:001> !rpcexts.eeinfo 0xb015f0
Computer Name: (null)
ProcessID: 708 (0x2C4)
System Time is: 3/21/2000 4:3:0:264
Generating component: 8
Status: 14
```

```
Detection Location: 311
```

```
Flags:
```

```
Parameter 0:(Long value) : -30976 (0xFFFF8700)
```

```
Parameter 1:(Long value) : 16777343 (0x100007F)
```

If the chain is very long and you wish to see only one record, use [!rpcexts.eerecord](#) instead.

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.eerecord

Article • 04/03/2024

The **!rpcexts.eerecord** extension displays the contents of an extended error information record.

```
dbgcmd
```

```
!rpcexts.eerecord EERecordAddress
```

Parameters

EERecordAddress

Specifies the address of the extended error record.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension displays the contents of one extended error information record in the debugger. In most cases, it is easier to use [!rpcexts.eeinfo](#), which displays the whole chain. If the chain is very long and you wish to see only one record, use [!eerecord](#) instead.

Here is an example:

```
dbgcmd
```

```
0:001> !rpcexts.eerecord 0xb015f0
Computer Name: (null)
ProcessID: 708 (0x2C4)
System Time is: 3/21/2000 4:3:0:264
Generating component: 8
Status: 14
```

Detection Location: 311

Flags:

Parameter 0:(Long value) : -30976 (0xFFFF8700)

Parameter 1:(Long value) : 16777343 (0x100007F)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!rpcexts.getcallinfo

Article • 04/03/2024

The **!rpcexts.getcallinfo** extension searches the system's RPC state information for server-side call (SCALL) information.

dbgcmd

```
!rpcexts.getcallinfo [ CallID | 0 [ IfStart | 0 [ ProcNum | 0xFFFF
[ProcessID|0] ] ] ]
!rpcexts.getcallinfo -?
```

Parameters

CallID

Specifies the call ID. This parameter is optional; include it if you only want to display calls matching a specific *CallID* value.

IfStart

Specifies the first DWORD of the interface UUID on which the call was made. This parameter is optional; include it if you only want to display calls matching a specific *IfStart* value.

ProcNum

Specifies the procedure number of this call. (The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file -- the first routine in the interface is 0, the second 1, and so on.)

ProcessID

Specifies the process ID (PID) of the server process that owns the calls you want to display. This parameter is optional; omit it if you want to display calls owned by multiple processes.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

The parameters are parsed from left to right. To skip a parameter, supply the value 0. There is one exception to this rule: the *ProcNum* parameter is skipped by supplying the value 0xFFFF.

Here is an example:

```
dbgcmd

0:002> !rpcexts.getcallinfo
Searching for call info ...
## PID  CELL ID    ST PNO IFSTART   TIDNUMBER CALLFLAG CALLID      LASTTIME
CONN/CLN
-----
00c4 0000.0006 00 009 367abb81 0000.0007 00000001 0000004a 00018b41
0000.0005
00c4 0000.000a 00 007 367abb81 0000.002d 00000001 0000009f 000134ff
0000.0009
00c4 0000.000d 00 00f 82273fdc 0000.002d 00000001 00000002 00036cd8
0000.0042
00c4 0000.0010 00 00f 367abb81 0000.002d 00000001 00000078 00011636
0000.000f
00c4 0000.0012 00 00d 8d9f4e40 0000.0007 00000001 0000004f 000097bd
0000.0011
00c4 0000.0015 00 000 367abb81 0000.0004 00000001 0000004c 0002cccf
0000.0014
00c4 0000.0017 00 007 367abb81 0000.0004 00000001 00000006 0000cf5e
0000.0016
00c4 0000.0018 00 000 367abb81 0000.002d 00000001 0000000b 0001236f
0000.002a
00c4 0000.0019 01 00b 82273fdc 0000.0002 00000009 00000000 00018b19
00d0.0104
00c4 0000.001b 00 009 65a93890 0000.0007 00000001 000000ea 0003cd14
0000.001a
00c4 0000.0021 00 03b 8d9f4e40 0000.0013 00000001 0000000b 0001162c
0000.0020
00c4 0000.0022 01 008 82273fdc 0000.001f 00000009 00000000 00013405
00c4.02e8
00c4 0000.0024 00 007 367abb81 0000.0004 00000001 00000006 0000f198
0000.0023
00c4 0000.0026 00 000 367abb81 0000.0036 00000001 000000ab 00038049
0000.0025
```

00c4 0000.0027 01 00b 82273fdc 0000.001f 00000009 00000000 00020b7c
00a8.0228
00c4 0000.0028 01 008 82273fdc 0000.003e 00000009 00000000 0003a949
0294.02f0
00c4 0000.0029 00 00d 8d9f4e40 0000.002d 00000001 0000033f 0003831a
0000.0031
00c4 0000.0030 00 03b 8d9f4e40 0000.0013 00000001 00000002 00024e43
0000.002f
00c4 0000.0032 01 008 82273fdc 0000.001f 00000009 00000000 000118f3
022c.019c
00c4 0000.0035 00 007 367abb81 0000.0033 00000001 00000074 0001042d
0000.0034
00c4 0000.0038 00 007 367abb81 0000.002d 00000001 0000000a 0002a3e4
0000.0037
00c4 0000.003a 00 007 367abb81 0000.0036 00000001 00000063 0003b7b8
0000.0039
00c4 0000.003b 00 004 3ba0ffc0 0000.002d 00000001 00000005 0002dd79
0000.002e
00c4 0000.003f 01 008 82273fdc 0000.0002 00000009 00000000 000245c6
01c0.037c
00c4 0000.0043 01 008 82273fdc 0000.0002 00000009 00000000 00037d50
020c.0394
00c4 0000.0049 00 008 8d9f4e40 0000.0007 00000001 000002b1 0004e900
0000.0048
0170 0000.0009 01 002 e60c73e6 0000.0002 00000009 baadf00d 0004ad30
020c.03a4
0170 0000.000a 01 002 0b0a6584 0000.0008 00000009 baadf00d 0001187b
00c4.012c
0170 0000.000c 01 002 0b0a6584 0000.0008 00000009 baadf00d 00011cdc
022c.019c
0170 0000.000d 01 003 00000136 0000.0011 00000009 baadf00d 00034845
020c.02b4
0170 0000.000e 01 000 412f241e 0000.0002 00000009 baadf00d 00012491
0294.02b8
0170 0000.000f 01 002 0b0a6584 0000.0011 00000009 baadf00d 000492e7
026c.0118
0170 0000.0010 01 002 e60c73e6 0000.0013 00000009 baadf00d 0004ab78
0378.038c
0170 0000.0014 01 004 e60c73e6 0000.0011 00000001 baadf00d 0002bc25
0378.024c
0170 0000.0015 01 003 00000136 0000.0013 00000009 00000003 00031d8d
0378.00b8
0170 0000.0018 01 004 00000136 0000.0002 00000001 baadf00d 00032e05
020c.026c
020c 0000.0004 01 003 00000132 0000.000b 00000009 00000000 00034953
0170.0240
020c 0000.000e 01 001 2f5f6520 0000.001e 00000009 00120006 00035bac
020c.03b4
020c 0000.0010 01 000 629b9f66 0000.000f 00000009 00000000 000279ff
00a8.0194
020c 0000.0011 01 004 faedcf59 0000.0003 00000009 00000012 0003836b
0378.024c
020c 0000.0012 01 001 629b9f66 0000.000f 00000009 00000000 0003657e
020c.02ec
020c 0000.0017 01 005 00000134 0000.0002 00000001 00000016 0003836b

```
0378.024c
020c 0000.001d 01 001 2f5f6520 0000.0014 00000001 0020007d 000351b2
020c.0258
0294 0000.0004 01 004 00000132 0000.0002 00000009 00000000 0003b786
0170.01ac
0378 0000.0004 01 003 00000134 0000.0003 0000000b 00300038 0002d896
020c.021c
026c 0000.0004 02 000 19bb5061 0000.0002 00000001 00000001 0004caa5
0000.0003
```

For a similar example using the DbgRpc tool, see [Get RPC Call Information](#).

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.getclientcallinfo

Article • 04/03/2024

The **!rpcexts.getclientcallinfo** extension searches the system's RPC state information for client call (CCALL) information.

dbgcmd

```
!rpcexts.getclientcallinfo [ CallID | 0 [ IfStart | 0 [ ProcNum | 0xFFFF  
[ProcessID|0] ] ] ]  
!rpcexts.getclientcallinfo -?
```

Parameters

CallID

Specifies the call ID. This parameter is optional; include it if you only want to display calls matching a specific *CallID* value.

IfStart

Specifies the first DWORD of the interface UUID on which the call was made. This parameter is optional; include it if you only want to display calls matching a specific *IfStart* value.

ProcNum

Specifies the procedure number of this call. (The RPC Run-Time identifies individual routines from an interface by numbering them by position in the IDL file -- the first routine in the interface is 0, the second 1, and so on.) This parameter is optional; include it if you only want to display calls matching a specific *ProcNum* value.

ProcessID

Specifies the process ID (PID) of the client process that owns the calls you want to display. This parameter is optional; omit it if you want to display calls owned by multiple processes.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension can only be used with CDB or with user-mode WinDbg. It is only available if full RPC state information is being gathered.

Here is an example:

```
dbgcmd

0:002> !rpcexts.getclientcallinfo
Searching for call info ...
## PID  CELL ID    PNO  IFSTART  TIDNUMBER CALLID    LASTTIME PS CLTNUMBER
ENDPOINT
-----
-- 
03d4 0000.0001 0000 19bb5061 0000.0000 00000001 0004ca9b 07 0000.0002 1118
```

For a similar example using the DbgRpc tool, see [Get RPC Client Call Information](#).

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.getdbgcell

Article • 04/03/2024

The `!rpcexts.getdbgcell` extension displays RPC state information for the specified cell.

dbgcmd

```
!rpcexts.getdbgcell ProcessID CellID1.CellID2  
!rpcexts.getdbgcell -?
```

Parameters

ProcessID

Specifies the process ID (PID) of the process whose server contains the desired cell.

CellID1.CellID2

Specifies the number of the cell to be displayed.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

dbgcmd

```
0:002> !rpcexts.getdbgcell c4 0.19  
Getting cell info ...  
Call
```

```
Status: Active
Procedure Number: 11
Interface UUID start (first DWORD only): 82273FDC
Call ID: 0x0 (0)
Servicing thread identifier: 0x0.3E
Call Flags: cached, LRPC
Last update time (in seconds since boot): 1453.459 (0x5AD.1CB)
Caller (PID/TID) is: d0.1ac (208.428)
```

For a similar example using the DbgRpc tool, see [Get RPC Cell Information](#).

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.getendpointinfo

Article • 04/03/2024

The `!rpcexts.getendpointinfo` extension searches the system's RPC state information for endpoint information.

dbgcmd

```
!rpcexts.getendpointinfo [EndpointName]
!rpcexts.getendpointinfo -?
```

Parameters

EndpointName

Specifies the number of the endpoint to be displayed. If omitted, the endpoints for all processes on the system are displayed.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

dbgcmd

```
0:002> !rpcexts.getendpointinfo
Searching for endpoint info ...
## PID  CELL ID  ST PROTSEQ      ENDPOINT
-----
```

00a8 0000.0001 01	NMP \PIPE\InitShutdown
00a8 0000.0003 01	NMP \PIPE\SfcApi
00a8 0000.0004 01	NMP \PIPE\ProfMapApi
00a8 0000.0005 01	LRPC OLE5
00a8 0000.0007 01	NMP \pipe\winlogonrpc
00c4 0000.0001 01	LRPC ntsvcs
00c4 0000.0003 01	NMP \PIPE\ntsvcs
00c4 0000.0008 01	NMP \PIPE\scerpc
00d0 0000.0001 01	NMP \PIPE\lsass
00d0 0000.0003 01	NMP \pipe\WMIEP_d0
00d0 0000.0006 01	LRPC policyagent
00d0 0000.0007 01	NMP \PIPE\POLICYAGENT
0170 0000.0001 01	LRPC epmapper
0170 0000.0003 01	TCP 135
0170 0000.0005 01	SPX 34280
0170 0000.0006 01	NB 135
0170 0000.0007 01	NB 135
0170 0000.000b 01	NMP \pipe\epmapper
01c0 0000.0001 01	NMP \pipe\spoolss
01c0 0000.0003 01	LRPC spoolss
01c0 0000.0007 01	LRPC OLE7
020c 0000.0001 01	LRPC OLE2
020c 0000.0005 01	LRPC senssvc
020c 0000.0007 01	NMP \pipe\tapsrv
020c 0000.0009 01	LRPC tapsrvlpc
020c 0000.000c 01	NMP \PIPE\ROUTER
020c 0000.0016 01	NMP \pipe\WMIEP_20c
0218 0000.0001 01	NMP \PIPE\winreg
022c 0000.0001 01	LRPC LRPC0000022c.00000001
022c 0000.0003 01	TCP 1041
022c 0000.0005 01	SPX 24576
022c 0000.0006 01	NMP \PIPE\atsvc
0294 0000.0001 01	LRPC OLE3
0378 0000.0001 01	LRPC OLE9
026c 0000.0001 01	TCP 1118
0344 0000.0001 01	LRPC OLE12

For a similar example using the DbgRpc tool, see [Get RPC Endpoint Information](#).

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.getthreadinfo

Article • 04/03/2024

The `!rpcexts.getthreadinfo` extension searches the system's RPC state information for thread information.

dbgcmd

```
!rpcexts.getthreadinfo ProcessID [ThreadID]
!rpcexts.getthreadinfo -?
```

Parameters

ProcessID

Specifies the process ID (PID) of the process containing the desired thread.

ThreadID

Specifies the thread ID of the thread to be displayed. If omitted, all threads in the specified process will be displayed.

-?

Displays some brief Help text for this extension in the Command Prompt window.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

dbgcmd

```
0:002> !rpcexts.getthreadinfo 26c
Searching for thread info ...
## PID  CELL ID  ST TID      LASTTIME
-----
026c 0000.0002 01 000003c4 0004caa5
026c 0000.0005 03 00000254 0004ca9b
```

For a similar example using the DbgRpc tool, see [Get RPC Thread Information](#).

Feedback

Was this page helpful?



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

!rpcexts.help

Article • 04/03/2024

The `!rpcexts.help` extension displays a Help text in the Command Prompt window showing all `Rpcexts.dll` extension commands.

```
dbgcmd
!rpcexts.help
```

DLL

`Rpcexts.dll`

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.rpcreadstack

Article • 04/03/2024

The `!rpcexts.rpcreadstack` extension reads an RPC client-side stack and retrieves the call information.

```
dbgcmd
```

```
!rpcexts.rpcreadstack ThreadStackPointer
```

Parameters

ThreadStackPointer

Specifies the pointer to the thread stack.

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

For a common use of this extension, see [Analyzing a Stuck Call Problem](#).

Here is an example:

```
dbgcmd
```

```
0:001> !rpcexts.rpcreadstack 68fba4
CallID: 1
IfStart: 19bb5061
ProcNum: 0
    Protocol Sequence:      "ncacn_ip_tcp"  (Address: 00692ED8)
    NetworkAddress: ""      (Address: 00692F38)
    Endpoint:           "1120"  (Address: 00693988)
```

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.rpctime

Article • 04/03/2024

The **!rpcexts.rpctime** extension displays the current system time.

```
dbgcmd
```

```
!rpcexts.rpctime
```

DLL

Rpcexts.dll

Remarks

This extension can only be used with CDB or with user-mode WinDbg.

Here is an example:

```
dbgcmd
```

```
0:001> !rpcexts.rpctime
Current time is: 059931.126 (0x00ea1b.07e)
```

Feedback

Was this page helpful?

 Yes

 No

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

!rpcexts.thread

Article • 04/03/2024

The **!rpcexts.thread** extension displays the per-thread RPC information.

This extension command should not be confused with the [.thread \(Set Register Context\)](#) command or the [!thread](#) (!kdextx86.thread and !kdexts.thread) extension.

```
dbgcmd
```

```
!rpcexts.thread TEB
```

Parameters

TEB

Specifies the address of the thread environment block (TEB).

DLL

Rpcexts.dll

Additional Information

For more information about debugging Microsoft Remote Procedure Call (RPC), see [RPC Debugging](#).

Remarks

This extension displays the per-thread RPC information. A field in the per-thread RPC information is the extended error information for this thread.

Here is an example:

```
dbgcmd
```

```
0:001> !rpcexts.thread 7ffdd000
RPC TLS at 692e70
```

```
HandleToThread - 0x6c
SavedProcedure - 0x0
SavedParameter - 0x0
```

```
ActiveCall - 0x0
Context - 0x0
CancelTimeout - 0xffffffff
SecurityContext - 0x0
ExtendedStatus - 0x0
ThreadEEInfo - 0xb015f0
ThreadEvent at - 0x00692E78
fCallCancelled - 0x0
buffer cache array at - 0x00692E84
fAsync - 0x0
```

Feedback

Was this page helpful?

 Yes

 No

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

ACPI Extensions (Acpikd.dll and Kdexts.dll)

Article • 10/25/2023

Extension commands that are useful for debugging ACPI (Advanced Configuration and Power Interface) BIOS code can be found in Acpikd.dll and Kdexts.dll.

Some of the ACPI debugging extensions can be found in Winxp\Acpikd.dll, while others can be found in Winxp\Kdexts.dll.

!acpicache

Article • 04/03/2024

The **!acpicache** extension displays all of the Advanced Configuration and Power Interface (ACPI) tables cached by the HAL.

dbgcmd

```
!acpicache [DisplayLevel]
```

Parameters

DisplayLevel

Specifies the detail level of the display. This value is either 0 for an abbreviated display or 1 for a more detailed display. The default value is 0.

DLL

Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see [ACPI Debugging](#) for information about other extensions that are associated with the ACPI.

Feedback

Was this page helpful?

 Yes

 No

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

!acpiinf

Article • 04/03/2024

The **!acpiinf** extension displays information on the configuration of the Advanced Configuration and Power Interface (ACPI), including the location of system tables and the contents of the ACPI fixed feature hardware.

```
dbgcmd
  !acpiinf
```

DLL

Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see [ACPI Debugging](#) for information about other extensions that are associated with the ACPI.

Feedback

Was this page helpful?

 Yes

 No

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

!acpiirqarb

Article • 04/03/2024

The **!acpiirqarb** extension displays the contents of the Advanced Configuration and Power Interface (ACPI) IRQ arbiter structure, which contains the configuration of I/O devices to system interrupt controller inputs and processor interrupt dispatch table (IDT) entries.

```
dbgcmd
```

```
!acpiirqarb
```

DLL

Kdexts.dll

Additional Information

For information about the ACPI, see the Microsoft Windows Driver Kit (WDK) documentation, the Windows SDK documentation, and *Microsoft Windows Internals* by Mark Russinovich and David Solomon. Also see [ACPI Debugging](#) for information about other extensions that are associated with the ACPI.

Feedback

Was this page helpful?

 Yes

 No

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

!acpikd.help

Article • 01/31/2024

The **!acpikd.help** extension displays help text in the Debugger Command window showing all Acpikd.dll extension commands.

```
dbgcmd
!acpikd.help
```

DLL

Acpikd.dll

Additional Information

For more information, see [ACPI Debugging](#).



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

!amli ?

Article • 01/31/2024

The **!amli ?** extension displays help text in the Debugger Command window for the **!amli** extension commands.

Syntax

```
dbgcmd
```

```
!amli ? [Command]
```

Parameters

Command

Specifies the **!amli** command whose help is to be displayed. For example, **!amli ? set** displays help for the **!amli set** command. If *Command* is omitted, a list of all commands is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!amli bc

Article • 10/25/2023

The **!amli bc** extension permanently clears an AML breakpoint.

Syntax

```
dbgcmd  
    !amli bc Breakpoint  
    !amli bc *
```

Parameters

Breakpoint Specifies the number of the breakpoint to be cleared.

* Specifies that all breakpoints should be cleared.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

To determine the breakpoint number of a breakpoint, use the [!amli bl](#) extension.

!amli bd

Article • 10/25/2023

The **!amli bd** extension temporarily disables an AML breakpoint.

Syntax

```
dbgcmd
```

```
!amli bd Breakpoint!amli bd *
```

Parameters

Breakpoint Specifies the number of the breakpoint to be disabled.

* Specifies that all breakpoints should be disabled.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

A disabled breakpoint can be re-enabled by using the [!amli be](#) extension.

To determine the breakpoint number of a breakpoint, use the [!amli bl](#) extension.

Here is an example of this command:

```
Console
```

```
kd> !amli bl
0: c29accf5 [\_WAK]
1: c29c20a5 [\_SB.PCI0.ISA.BAT1._BST]

kd> !amli bd 1
```

!amli be

Article • 10/25/2023

The **!amli be** extension enables an AML breakpoint.

Syntax

```
dbgcmd  
    !amli be Breakpoint!amli be *
```

Parameters

Breakpoint Specifies the breakpoint number of the breakpoint to be enabled.

* Specifies that all breakpoints should be enabled.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

All breakpoints are enabled when they are created. Breakpoints are only disabled if you have used the [!amli bd](#) extension.

To determine the breakpoint number of a breakpoint, use the [!amli bl](#) extension.

!amli bl

Article • 01/31/2024

The **!amli bl** extension displays a list of all AML breakpoints.

Syntax

```
dbgcmd
  !amli bl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The amli Debugger](#).

Remarks

The !amli Debugger supports a maximum of ten breakpoints.

Here is an example of the **!amli bl** extension:

```
Console
kd> !amli bl
0: <e> ffffffff80e5e2f1:[\_SB.LNKD._SRS]
1: <e> ffffffff80e5d969:[\_SB.LNKB._STA]
2: <d> ffffffff80e630c9:[\_WAK]
3: <e> ffffffff80e612c9:[\_SB.MBRD._CRS]
```

The first column gives the breakpoint number. The **<e>** and **<d>** marks indicate whether the breakpoint is enabled or disabled. The address of the breakpoint is in the next column. Finally, the method containing the breakpoint is listed, with the offset of the breakpoint if it is not set at the start of the method.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

!amli bp

Article • 10/25/2023

The **!amli bp** extension places a breakpoint in AML code.

Syntax

```
dbgcmd
  !amli bp { MethodName | CodeAddress }
```

Parameters

MethodName

Specifies the full path of the method name on which the breakpoint will be set.

CodeAddress

Specifies the address of the AML code at which the breakpoint will be set. If *CodeAddress* is prefixed with two percent signs (%%), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

The AMLI Debugger supports a maximum of 10 breakpoints.

Here is an example. The following command will set a breakpoint on the _DCK method:

```
Console
  kd> !amli bp \_sb.pci0.dock._dck
```

!amlcli

Article • 10/25/2023

The **!amlcli** extension clears the AML interpreter's event log.

Syntax

```
dbgcmd
```

```
!amlcli cl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

!amli debugger

Article • 10/25/2023

The **!amli debugger** extension breaks into the AMLI Debugger.

Syntax

```
dbgcmd
```

```
!amli debugger
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

When this command is issued, notification is sent to the AML interpreter. The next time the interpreter is active, it will immediately break into the AMLI Debugger.

The **!amli debugger** extension only causes one break. If you want it to break again, you need to use this extension again, or set a breakpoint.

!amli dh

Article • 10/25/2023

The **!amli dh** extension displays the AML interpreter's internal heap block.

Syntax

dbgcmd

```
!amli dh [HeapAddress]
```

Parameters

HeapAddress

Specifies the address of the heap block. If this is omitted, the global heap is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

!amli dl

Article • 10/25/2023

The **!amli dl** extension displays a portion of the AML interpreter's event log.

Syntax

```
dbgcmd
```

```
!amli dl
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

The event log chronicles the most recent 150 events that occurred in the interpreter.

Here is an example of the log display:

```
Console
```

```
kd> !amli dl
RUN!: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000000: Ctx=c18b4000,rc=0
KICK: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000000: rc=0
SYNC:
[c15a6618]QTh=00000000,QCt=00000000,QFg=00000002,LockPhase=0,Locked=0,IRQL=0
0: Obj=\_WAK
ASYN:
[c15a6618]QTh=00000000,QCt=00000000,QFg=00000002,LockPhase=0,Locked=0,IRQL=0
0: Obj=\_WAK
REST: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002:
Ctx=c18b4000,Obj=\_WAK
INSQ: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002:
Ctx=c18b4000,Obj=\_WAK
EVAL: [c15a6618]QTh=00000000,QCt=00000000,QFg=00000002:
Ctx=c18b4000,Obj=\_WAK
RUNC: [c15a6618]QTh=c15a6618,QCt=c18b4000,QFg=00000002:
Ctx=c18b4000,Obj=\_WAK
```

!amli dns

Article • 10/25/2023

The **!amli dns** extension displays an ACPI namespace object.

Syntax

dbgcmd

```
!amli dns [/s] [Name | Address]
```

Parameters

/s

Causes the entire namespace subtree under the specified object to be displayed recursively.

Name

Specifies the namespace path.

Address

Specifies the address of the namespace node.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

If neither *Name* nor *Address* is specified, the entire ACPI namespace tree is displayed recursively. The */s* parameter is always assumed in this case, even if it is not specified.

This command is useful for determining what a particular namespace object is—whether it is a method, a field unit, a device, or another type of object.

Without the */s* parameter, this extension is equivalent to the [!nsobj](#) extension. With the */s* parameter, it is equivalent to the [!nstree](#) extension.

Here are some examples. The following command displays the namespace for the object bios:

```
Console

AMLI(? for help)-> dns \bios

ACPI Name Space: \BIOS (80E5F378)
OpRegion(BIOS:RegionSpace=SystemMemory,Offset=0xfc07500,Len=2816)
```

The following command displays the namespace for the object _BST, and the tree subordinate to it:

```
Console

kd> !amli dns /s \_sb.pci0.isa.bat1._bst

ACPI Name Space: \_SB.PCI0.ISA.BAT1._BST (c29c2044)
Method(_BST:Flags=0x0,CodeBuff=c29c20a5,Len=103)
```

To display the namespace for the device BAT1, type:

```
Console

kd> !amli dns /s \_sb.pci0.isa.bat1
```

To display the namespace of everything subordinate to the DOCK device, type:

```
Console

kd> !amli dns /s \_sb.pci0.dock
```

To display the namespace subordinate to the _DCK method, type:

```
Console

kd> !amli dns /s \_sb.pci0.dock._dck
```

To display the entire namespace, type:

```
Console

kd> !amli dns
```

!aml do

Article • 10/25/2023

The **!aml do** extension displays an AML data object.

Syntax

```
dbgcmd
```

```
!aml do Address
```

Parameters

Address

Specifies the address of the data object.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

!amli ds

Article • 01/31/2024

The **!amli ds** extension displays an AML stack.

Syntax

```
dbgcmd
```

```
!amli ds [/v] [Address]
```

Parameters

/v

Causes the display to be verbose.

Address

Specifies the address of the context block whose stack is desired. If *Address* is omitted, the current context is used.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



Windows driver documentation feedback

Windows driver documentation is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

!amli find

Article • 10/25/2023

The **!amli find** extension finds an ACPI namespace object.

Syntax

```
dbgcmd
```

```
!amli find Name
```

Parameters

Name

Specifies the name of the namespace object (without the path).

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

The **!amli find** command takes the name of the object and returns the full path and name. The *Name* parameter must be the final segment of the full path and name.

Here are some examples. The following command will find all declarations of the object _SRS:

```
Console
```

```
kd> !amli find _srs
\_SB.LNKA._SRS
\_SB.LNKB._SRS
\_SB.LNKC._SRS
\_SB.LNKD._SRS
```

This is not simply a text search. The command `!amlfi find srs` does not display any hits, because the final segment of each of these declarations is "_SRS", not "SRS". The command `!amlfi find LNK` similarly does not return hits. The command `!amlfi find LNKB` would display the single node that terminates in "LNKB", not the four children of this node shown in the previous display:

Console

```
kd> !amlfi find lnkb
\_SB.LNKB.
```

If you need to see the children of a node, use the `!amlfi dns` command with the `/s` parameter.

Here is another example, issued from the AMLI Debugger prompt. This shows all declarations of the object _BST in the namespace:

Console

```
AMLI(? for help)-> find _bst
\_SB.PCI0.ISA.BAT1._BST
\_SB.PCI0.ISA.BAT2._BST
```

!amli lc

Article • 10/25/2023

The **!amli lc** extension lists all active ACPI contexts.

Syntax

```
dbgcmd
```

```
!amli lc
```

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

Each context corresponds to a method currently running in the AML interpreter.

Here is an example:

```
Console
```

```
AMLI(? for help)-> lc
  Ctxt=80e3f000, ThID=00000000, Flgs=A--C-----, pb0p=00000000,
  Obj=\_SB.LNKA._STA
  Ctxt=80e41000, ThID=00000000, Flgs=A--C-----, pb0p=00000000,
  Obj=\_SB.LNKB._STA
  Ctxt=80e9a000, ThID=00000000, Flgs=A--C-----, pb0p=00000000,
  Obj=\_SB.LNKC._STA
  Ctxt=80ea8000, ThID=00000000, Flgs=A--C-----, pb0p=00000000,
  Obj=\_SB.LNKD._STA
  *Ctxt=80e12000, ThID=80e6eda8, Flgs=---CR----, pb0p=80e5d5ac,
  Obj=\_SB.LNKA._STA
```

The **Obj** field gives the full path and name of the method as it appears in the ACPI tables.

The **Ctxt** field gives the address of the context block. The asterisk (*) indicates the *current context*. This is the context that was being executed by the interpreter when the break occurred.

The abbreviation **pbOp** indicates the instruction pointer (pointer to binary op codes).

There are nine flags that can be displayed in the **Flgs** section. If a flag is not set, a hyphen is displayed instead. The full list of flags is as follows:

Flag	Meaning
A	Asynchronous evaluation
N	Nested evaluation
Q	In the ready queue
C	Needs a callback
R	Running
W	Ready
T	Time-out
D	Timer dispatch
P	Timer pending

!amli In

Article • 10/25/2023

The **!amli In** extension displays the specified method or the method containing a given address.

Syntax

dbgcmd

```
!amli In [ MethodName | CodeAddress ]
```

Parameters

MethodName

Specifies the full path of the method name. If *MethodName* specifies an object that is not actually a method, an error results.

CodeAddress

Specifies the address of the AML code that is contained in the desired method. If *CodeAddress* is prefixed with two percent signs (%%), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

If neither *MethodName* nor *CodeAddress* is specified, the method associated with the current context is displayed.

The following command shows the method being currently run:

Console

```
kd> !amli ln  
c29accf5: \_WAK
```

The method _WAK is shown, with address 0xC29ACCF5.

!amli r

Article • 10/25/2023

The **!amli r** extension displays information about the current context or the specified context.

Syntax

dbgcmd

```
!amli r [ContextAddress]
```

Parameters

ContextAddress

Specifies the address of the context block to be displayed. The address of a context block can be determined from the **Ctxt** field in the **!amli lc** display. If *ContextAddress* is prefixed with two percent signs (%%), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address. If this parameter is omitted, the current context is displayed.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

If the AMLI Debugger prompt appears suddenly, this is a useful command to use.

For example, the following command will display the current context of the interpreter:

Console

```
AMLI(? for help)-> r  
  
Context=c18b4000*, Queue=00000000, ResList=00000000  
ThreadID=c15a6618, Flags=00000010
```

```
StackTop=c18b5eec, UsedStackSize=276 bytes, FreeStackSize=7636 bytes
LocalHeap=c18b40c0, CurrentHeap=c18b40c0, UsedHeapSize=88 bytes
Object=\_WAK, Scope=\_WAK, ObjectOwner=c18b4108, SyncLevel=0
AsyncCallBack=ff06b5d0, CallBackData=0, CallBackContext=c99efddc
```

```
MethodObject=\_WAK
80e0ff5c: Local0=Unknown()
80e0ff70: Local1=Unknown()
80e0ff84: Local2=Unknown()
80e0ff98: Local3=Unknown()
80e0ffac: Local4=Unknown()
80e0ffc0: Local5=Unknown()
80e0ffd4: Local6=Unknown()
80e0ffe8: Local7=Unknown()
80e0e040: RetObj=Unknown()
```

```
Next AML Pointer: ffffffff80e630df:[\_WAK+16]
```

```
fffffff80e630df : If(S4BW
fffffff80e630e5 : {
fffffff80e630e5 : | Store(Zero, S4BW)
fffffff80e630eb : }
```

!amliset

Article • 10/25/2023

The **!amliset** extension sets or displays the AMLI Debugger options.

dbgcmd

!amliset Options

Parameters

Options Specifies one or more options to be set. Separate multiple options with spaces. Possible values include:

spewon

Causes full debug output to be sent from the target computer. This option should be left on at all times for effective AML debugging. See the Remarks section for details.

spewoff

Suppresses debug output.

verboseon

Turns on verbose mode. This causes the AMLI Debugger to display the names of AML methods as they are evaluated.

verboseoff

Turns off verbose mode.

traceon

Activates ACPI tracing. This produces much more output than the **verboseon** option. This option is very useful for tracking SMI-related hard hangs.

traceoff

Deactivates ACPI tracing.

nesttraceon

Activates nest tracing. This option is only effective if the **traceon** option is also selected.

dbgbrkon

Enables breaking into the AMLI Debugger.

dbgbrkoff

Deactivates the **dbgbrkon** option.

nesttraceoff

Deactivates nest tracing.

Ibrkon

Breaks into the AMLI Debugger when DDB loading is completed.

Ibrkoff

Deactivates the **Ibrkon** option.

errbrkon

Breaks into the AMLI Debugger whenever the interpreter has a problem evaluating AML code.

errbrkoff

Deactivates the **errbrkon** option.

logon

Enables event logging.

logoff

Disables event logging.

logmuton

Enables mutex event logging.

logmutoff

Disables mutex event logging.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

If no options are specified, the current status of all options is displayed.

By default, many messages are filtered out, you may need to turn this output on with **!amliset spewon**. Otherwise, numerous AMLI Debugger messages will be lost.

If the AML interpreter breaks into the AMLI Debugger, this output will be automatically turned on.

For more details on this output filtering, see **DbgPrintEx** and **KdPrintEx** in the Windows Driver Kit (WDK) documentation.

!amli u

Article • 10/25/2023

The **!amli u** extension unassembles AML code.

Syntax

dbgcmd

```
!amli u [ MethodName | CodeAddress ]
```

Parameters

MethodName

Specifies the full path of the method name to be disassembled.

CodeAddress

Specifies the address of the AML code where disassembly will begin. If *CodeAddress* is prefixed with two percent signs (%%), it is interpreted as a physical address. Otherwise, it is interpreted as a virtual address.

DLL

Kdexts.dll

Additional Information

For information about related commands and their uses, see [The AMLI Debugger](#).

Remarks

If neither *MethodName* nor *CodeAddress* is specified and you are issuing this command from an AMLI

The disassembly display will continue until the end of the method is reached.

Note The standard [u \(Unassemble\)](#) command will not give proper results with AML code.

Here are some examples. To disassemble the object at address 0x80E5D701, use the following command:

Console

```
kd> !amli u 80e5d701

ffffffff80e5d701 : CreateWordField(CRES, 0x1, IRQW)
ffffffff80e5d70c : And(\_SB_.PCI0.LPC_.PIRA, 0xf, Local0)
ffffffff80e5d723 : Store(One, Local1)
ffffffff80e5d726 : ShiftLeft(Local1, Local0, IRQW)
ffffffff80e5d72d : Return(CRES)
```

The following command will disassemble the _DCK method:

Console

```
kd> u \_sb.pci0.dock._dck
```

!facs

Article • 10/25/2023

The **!facs** extension displays a Firmware ACPI Control Structure (FACS).

Syntax

```
dbgcmd
```

```
!facs Address
```

Parameters

Address

Specifies the address of the FACS.

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

!fadt

Article • 10/25/2023

The **!fadt** extension displays a Fixed ACPI Description Table (FADT).

Syntax

```
dbgcmd
```

```
!fadt Address
```

Parameters

Address

Specifies the address of the FADT.

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

!mapic

Article • 10/25/2023

The **!mapic** extension displays an ACPI Multiple APIC table.

Syntax

```
dbgcmd
```

```
!mapic Address
```

Parameters

Address

Specifies the address of the Multiple APIC Table.

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

!nsobj

Article • 10/25/2023

The **!nsobj** extension displays an ACPI namespace object.

Syntax

dbgcmd

```
!nsobj [Address]
```

Parameters

Address

Specifies the address of the namespace object. If this is omitted, the root of the namespace tree is used.

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

Remarks

This extension is equivalent to [!amli dns](#).

!nstree

Article • 10/25/2023

The **!nstree** extension displays an ACPI namespace object and its children in the namespace tree.

Syntax

dbgcmd

```
!nstree [Address]
```

Parameters

Address

Specifies the address of the namespace object. This object and the entire namespace tree subordinate to it will be displayed. If *Address* is omitted, the entire namespace tree is displayed.

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

Remarks

This extension is equivalent to [!amli dns /s](#).

!rsdt

Article • 10/25/2023

The **!rsdt** extension displays the ACPI Root System Description Table.

Syntax

```
dbgcmd
```

```
!rsdt
```

DLL

Kdexts.dll

Additional Information

For more information, see [ACPI Debugging](#).

Kernel Streaming Extensions (Ks.dll)

Article • 10/25/2023

Extension commands that are useful for debugging kernel streaming drivers and AVStream drivers can be found in Ks.dll.

Most of the sample output in these reference pages was generated by debugging the filter-centric sample Avssamp.sys. This sample is included in the Windows Driver Kit.

You need special symbols to use this extension.

You can get additional information for many of the extension commands in this section simply by entering the command into the debugger with no arguments.

For more information, see [Kernel Streaming Debugging](#).

!ks.help

Article • 10/25/2023

The **!ks.help** extension displays a help text showing all AVStream-specific Ks.dll extension commands.

```
dbgcmd
```

```
!ks.help
```

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

!ks.kshelp

Article • 10/25/2023

The **!ks.kshelp** extension displays a help text showing original KS 1.0-specific Ks.dll extension commands.

dbgcmd

!ks.kshelp

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

!ks.pchelp

Article • 10/25/2023

The **!ks.pchelp** extension displays a help text showing PortCls-specific Ks.dll extension commands.

```
dbgcmd
```

```
!ks.pchelp
```

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

!ks.allstreams

Article • 10/25/2023

The **!ks.allstreams** extension walks the entire device tree and finds every kernel streaming device in the system.

dbgcmd

```
!ks.allstreams [Flags] [Level]
```

Parameters

Flags

Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x1:

Bit 0 (0x1)

Causes the display to include streams.

Bit 2 (0x4)

Causes the display to include proxy instances.

Bit 3 (0x8)

Causes the display to include queued IRPs.

Bit 4 (0x10)

Causes the display to include an unformatted display of all streams.

Bit 5 (0x20)

Causes the display to include an unformatted display of all stream formats.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

This command can take some time to execute (a minute is not unusual).

Here is an example of the `!ks.allstreams` display:

```
dbgcmd

kd> !allstreams
6 Kernel Streaming FDOs found:
  Functional Device 82a17690 [\Driver\smwdm]
  Functional Device 8296eb08 [\Driver\wdmaud]
  Functional Device 82490388 [\Driver\sysaudio]
  Functional Device 82970cb8 [\Driver\MSPQM]
  Functional Device 824661b8 [\Driver\MSPCLOCK]
  Functional Device 8241c020 [\Driver\avssamp]
```

!ks.automation

Article • 10/25/2023

The **!ks.automation** extension displays any automation items associated with the given object.

```
dbgcmd
```

```
!ks.automation Object
```

Parameters

Object

Specifies a pointer to the object for which to display automation items. (Automation items are properties, methods, and events.) *Object* must be one of the following types: PKSPIN, PKSFILTER, CKsPin*, CKsFilter*, PIRP. If *Object* is a pointer to an automation IRP, the command returns property information and handlers.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

You can use this command with a filter address obtained from [!ks.enumdevobj](#).

Here is an example of the **!ks.automation** display. The argument is the address of a filter:

```
dbgcmd
```

```
kd> !automation 829493c4
Filter 829493c4 has the following automation items:
  Property Items:
    Set KSPROPSETID_Pin
```

```
Item ID = KSPROPERTY_PIN_CINSTANCES
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000008
Item ID = KSPROPERTY_PIN_CTYPES
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000018
    MinData = 00000004
Item ID = KSPROPERTY_PIN_DATAFLOW
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000004
Item ID = KSPROPERTY_PIN_DATARANGES
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000000
Item ID = KSPROPERTY_PIN_DATAINTERSECTION
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000028
    MinData = 00000000
Item ID = KSPROPERTY_PIN_INTERFACES
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000000
Item ID = KSPROPERTY_PIN_MEDIUMS
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000000
Item ID = KSPROPERTY_PIN_COMMUNICATION
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000004
Item ID = KSPROPERTY_PIN_NECESSARYINSTANCES
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000004
Item ID = KSPROPERTY_PIN_CATEGORY
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000010
Item ID = KSPROPERTY_PIN_NAME
    Get Handler = ks!CKsFilter::Property_Pin
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000000
```

```

Set KSPROPSETID_Topo
Item ID = KSPROPERTY_TOPOLOGY_CATEGORIES
    Get Handler = ks!CKsFilter::Property_Topo
    Set Handler = NULL
    MinProperty = 00000018
    MinData = 00000000
Item ID = KSPROPERTY_TOPOLOGY_NODES
    Get Handler = ks!CKsFilter::Property_Topo
    Set Handler = NULL
    MinProperty = 00000018
    MinData = 00000000
Item ID = KSPROPERTY_TOPOLOGY_CONNECTIONS
    Get Handler = ks!CKsFilter::Property_Topo
    Set Handler = NULL
    MinProperty = 00000018
    MinData = 00000000
Item ID = KSPROPERTY_TOPOLOGY_NAME
    Get Handler = ks!CKsFilter::Property_Topo
    Set Handler = NULL
    MinProperty = 00000020
    MinData = 00000000
Set KSPROPSETID_General
Item ID = KSPROPERTY_GENERAL_COMPONENTID
    Get Handler = ks!CKsFilter::Property_General_ComponentId
    Set Handler = NULL
    MinProperty = 00000018
    MinData = 00000048
Set [ks!KSPROPSETID_Frame] a60d8368-5324-4893-b020-c431a50bcbe3
Item ID = 0
    Get Handler = ks!CKsFilter::Property_Frame_Holding
    Set Handler = ks!CKsFilter::Property_Frame_Holding
    MinProperty = 00000018
    MinData = 00000004
Method Items:
    NO SETS FOUND!
Event Items:
    NO SETS FOUND!

```

!ks.devhdr

Article • 10/25/2023

The **!ks.devhdr** extension displays the kernel streaming device header associated with the given WDM object.

dbgcmd

```
!ks.devhdr DeviceObject
```

Parameters

DeviceObject

This parameter specifies a pointer to a WDM device object. If *DeviceObject* is not valid, the command returns an error.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The output from [!ks.allstreams](#) can be used as the input for **!ks.devhdr**.

Here is an example of the **!ks.devhdr** display:

dbgcmd

```
kd> !devhdr 827aedf0 7
Device Header 824ca1e0
    Child Create Handler List:
        Create Item eb3a7284
        CreateFunction =
sysaudio!CFilterInstance::FilterDispatchCreate+0x00
```

```
ObjectClass = NULL  
Flags = 0
```

!ks.dump

Article • 10/25/2023

The **!ks.dump** extension displays the specified object.

dbgcmd

```
!ks.dump Object [Level] [Flags]
```

Parameters

Object

Specifies a pointer to an AVStream structure, an AVStream class object, or a PortCls object. Can also specify a pointer to an IRP or a file object.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7. You can see more information about levels by issuing a **!ks.dump** command with no arguments.

Flags

Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)

Display all queued IRPs.

Bit 1 (0x2)

Display all pending IRPs.

Bit 2 (0x4)

Analyze a stalled graph for suspects.

Bit 3 (0x8)

Show all pin states.

DLL

Windows 2000

winxp\Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The **!ks.dump** command recognizes most AVStream objects, including pins, filters, factories, devices, pipes, and stream pointers. This command also recognizes some stream class structures, including stream objects, filter instances, device extensions, and SRBs.

Following is an example of the **!ks.dump** display for a filter:

```
dbgcmd  
  
kd> !dump 829493c4  
Filter object 829493c4 [CKsFilter = 82949350]  
    Descriptor      f7a233c8:  
    Context        829dce28
```

Following is an example of the **!ks.dump** display for a pin:

```
dbgcmd  
  
kd> !dump 8160DDE0 7  
Pin object 8160DDE0 [CKsPin = 8160DD50]  
    DeviceState    KSSTATE_RUN  
    ClientState    KSSTATE_RUN  
    ResetState     KSRESET_END  
    CKsPin object 8160DD50 [KSPIN = 8160DDE0]  
        State          KSSTATE_RUN  
        Processing Mutex 8160DFD0 is not held  
        And Gate &      8160DF88  
        And Gate Count  1
```

Some important parts of this display are included in the following table.

Parameter	Meaning
DeviceState	The state that the pin was requested to enter. If different from ClientState, this is the state that the minidriver will transition to next.

ClientState	The state that the minidriver is actually in. This reflects the state of the pipe.
ResetState	Indicates whether or not the object is in the middle of a flush. KSRESET_BEGIN indicates a flush. KSRESET_END indicates no flush.
State	The internal state of the pin's transport to non-AVStream filters.

Following is an example of the **!ks.dump** display for a stream class driver:

```
dbgcmd

kd> !dump 81a0a170 7
Device Extension 81a0a228:
  Device Object          81a0a170 [\Driver\TESTCAP]
  Next Device Object     81bd56d8 [\Driver\PnpManager]
  Physical Device Object 81bd56d8 [\Driver\PnpManager]
  REGISTRY FLAGS:
    Page out driver when closed
    No suspend if running
  MINIDRIVER Data:
    Device Extension      81a0a44c
    Interrupt Routine     00000000
    Synchronize Routine   STREAM!StreamClassSynchronizeExecution
    Receive Device SRB    testcap!AdapterReceivePacket
    Cancel Packet          testcap!AdapterCancelPacket
    Timeout Packet         testcap!AdapterTimeoutPacket
    Size (d / r / s / f)  1a0(416), 14(20), 978(2424), 0(0)
    Sync Mode              Driver Synchronizes
  Filter Type 0:
    Symbolic Links:
      Information Paged Out
    Instances:
      816b7bd8
```

Note that the sizes are listed both in hexadecimal numbers, and then, parenthetically in the decimal equivalent. The Size abbreviations in this display are listed in the following table.

Size	Explanation
d	Device
r	Request

s	Stream
f	Filter. If the filter size is 0, the filter is single instance. If it is greater than 0, it is multi-instance.

!ks.dumpbag

Article • 10/25/2023

The **!ks.dumpbag** extension displays the contents of the object bag for the specified object.

dbgcmd

```
!ks.dumpbag Object [Level]
```

Parameters

Object

Specifies a pointer to a valid client viewable object structure, or to the private class object.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Here is an example of the **!ks.dumpbag** display for a filter:

dbgcmd

```
kd> !dumpbag 829493c4
Filter 829493c4 [CKsFilter = 82949350]:
    Object Bag 829493d0:
        Object Bag Item 829dce28:
```

Reference Count : 1
Item Cleanup Handler : f7a21730

!ks.dumpcircuit

Article • 10/25/2023

The **!ks.dumpcircuit** extension lists details of the transport circuit associated with the given object.

dbgcmd

```
!ks.dumpcircuit [Object] [Level]
```

Parameters

Object

Specifies a pointer to the object for which to display the transport circuit. For AVStream, *Object* must be one of the following types: CKsPin*, CKsQueue*, CKsRequestor*, CKsSplitter*, CKsSplitterBranch*.

For PortCls, *object* must be one of the following types: CPortPin*, CKsShellRequestor*, or ClrpStream*.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Note that **!ks.dumpcircuit** starts walking the circuit at the specified object, which does not always correspond to the data source.

You can first use **!ks.graph** with a filter address to list pin addresses, and then use these addresses with **!ks.dumpcircuit**.

Here is an example of the **!ks.dumpcircuit** display:

```
dbgcmd  
  
kd> !dumpcircuit 8293f4f0  
Pin8293f4f0 0 (snk, out)  
Queue82990e20 r/w/c=2489/2/0
```

!ks.dumplog

Article • 10/25/2023

The **!ks.dumplog** extension displays the internal kernel streaming debug log.

dbgcmd

!ks.dumplog [Entries]

Parameters

Entries

Optional. Specifies the number of log entries to display. If *Entries* is zero or omitted, the entire log is displayed.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

You can stop the log display by pressing **CTRL+C**.

This extension requires that the target computer be running a checked (debug) version of Ks.sys.

!ks.dumpqueue

Article • 10/25/2023

The **!ks.dumpqueue** extension displays information about the queues associated with a given AVStream object, or the stream associated with a port class object.

dbgcmd

```
!ks.dumpqueue Object [Level]
```

Parameters

Object

Specifies a pointer to the object for which to display the queue. *Object* must be of type PKSPIN, PKSFILTER, CKsPin*, CKsFilter*, CKsQueue*, CPortPin*, or CPortFilter*.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Object must be a filter or a pin. For a pin, a single queue is displayed. For a filter, multiple queues are displayed.

This command can take some time to execute.

Here is an example of the **!ks.dumpqueue** display:

dbgcmd

```
kd> !dumpqueue 829493c4
Filter 829493c4: Output Queue 82990e20:
Queue 82990e20:
    Frames Received : 1889
    Frames Waiting : 3
    Frames Cancelled : 0
    And Gate 82949464 : count = 1, next = 00000000
    Frame Gate NULL
    Frame Header 82aaef78:
NextFrameHeaderInIrp = 00000000
    OriginalIrp = 82169e48
    Mdl = 8292e358
    Irp = 82169e48
StreamHeader = 8298dea0
    FrameBuffer = edba3000
    StreamHeaderSize = 00000000
    FrameBufferSize = 00025800
    Context = 00000000
    Refcount = 1
```

!ks.enumdevobj

Article • 10/25/2023

The **!ks.enumdevobj** extension displays the KSDEVICE associated with a given WDM device object, and lists the filter types and filters currently instantiated on this device.

dbgcmd

```
!ks.enumdevobj DeviceObject
```

Parameters

DeviceObject

Specifies a pointer to a WDM device object.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The output from [!ks.allstreams](#) can be used as the input for **!ks.enumdevobj**.

Here is an example of the **!ks.enumdevobj** display:

dbgcmd

```
kd> !enumdevobj 8241c020
WDM device object 8241c020:
    Corresponding KSDEVICE      823b8430
    Factory 829782dc [Descriptor f7a233c8] instances:
        829493c4
```

!ks.enumdrvobj

Article • 10/25/2023

The **!ks.enumdrvobj** extension displays all KSDEVICE structures associated with a given WDM driver object, and lists the filter types and filters currently instantiated on these devices.

dbgcmd

```
!ks.enumdrvobj DriverObject
```

Parameters

DriverObject

Specifies a pointer to a WDM driver object.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Since **!ks.enumdrvobj** enumerates every device chained off a WDM driver object, it is equivalent to invoking [!ks.enumdevobj](#) on every device chained off a given driver.

!ks.eval

Article • 10/25/2023

The **!ks.eval** extension evaluates an expression using an extension-specific expression evaluator.

dbgcmd

!ks.eval Expression

Parameters

Expression

Specifies the expression to evaluate. *Expression* can include any MASM operators, symbols, or numerical syntax, as well as the extension-specific operators described below. For more information about MASM expressions, see [MASM Numbers and Operators](#).

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The extension module includes two extension-specific operators which can be used in address parameters to extension commands:

fdo(x)

Returns the functional device object associated with the object at address **x**.

driver(x)

Returns the driver object associated with **fdo(x)**.

You can use the **!ks.eval** command to parse expressions that contain these extension-specific operators as well as [MASM Numbers and Operators](#).

Note that all operators supported by **!ks.eval** are also supported by all other extension commands in the Ks.dll extension module.

Here is an example of the **!ks.eval** extension being used with the address of a filter. Note the presence of the 0x8241C020 address in the [**!ks.allstreams**](#) output:

```
dbgcmd

kd> !eval fdo(829493c4)
Resulting Evaluation: 8241c020

kd> !allstreams
6 Kernel Streaming FDOs found:
    Functional Device 82a17690 [\Driver\smwdm]
    Functional Device 8296eb08 [\Driver\wdmaud]
    Functional Device 82490388 [\Driver\sysaudio]
    Functional Device 82970cb8 [\Driver\MSPQM]
    Functional Device 824661b8 [\Driver\MSPCLOCK]
    Functional Device 8241c020 [\Driver\avssamp]
```

!ks.findlive

Article • 10/25/2023

The **!ks.findlive** extension searches the internal Ks.sys debug log to attempt to find live objects of a specified type.

dbgcmd

```
!ks.findlive Type [Entries] [Level]
```

Parameters

Type

Specifies the type of object for which to search. Enter one of the following as a literal value: **Queue**, **Requestor**, **Pin**, **Filter**, or **Irp**.

Entries

If this parameter is zero or omitted, the entire log is searched.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The **!ks.findlive** command may not find all possible specified live objects.

This extension requires that the target computer be running a checked (debug) version of Ks.sys.

!ks.forcedump

Article • 10/25/2023

The **!ks.forcedump** command displays information about memory contents at a caller-supplied address.

dbgcmd

```
!ks.forcedump Object Type [Level]
```

Parameters

Object

Specifies a pointer to the object for which to display information.

Type

Specifies the type of object.

For AVStream/KS objects, *Type* must be one of the following values: CKsQueue, CKsDevice, CKsFilterFactory, CKsFilter, CKsPin, CKsRequestor, CKsSplitter, CKsSplitterBranch, CKsPipeSection, KSPIN, KSFILTER, KSFILTERFACTORY, KSDEVICE, KSSTREAM_POINTER, KSPFRAME_HEADER, KSIOBJECT_HEADER, KSPDO_EXTENSION, KSIDevice_HEADER, KSSTREAM_HEADER, KSPIN_DESCRIPTOR_EX, CKsProxy, CKsInputPin, CKsOutputPin, CasyncItemHandler.

For Port Class objects, *Type* must be one of the following values: DEVICE_CONTEXT, CPortWaveCyclic, CPortPinWaveCyclic, CPortTopology, CPortDMus, ClrpStream, CKsShellRequestor, CPortFilterWaveCyclic, CDmaChannel, CPortWavePci, CportPinWavePci.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

DLL

Windows 2000

winxp\Ks.dll

Windows XP and later

Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Normally, you can use [!ks.dump](#) to display data structures.

However, if symbols are loaded incorrectly or too much information is paged out, the type identification logic in the [!ks.dump](#) command may fail to identify the type of structure at a given address.

If this happens, try using the [!ks.forcedump](#) command. This command works just like [!ks.dump](#) except that the user specifies the type of the object.

Note The [!ks.forcedump](#) command does not verify that *Type* is the correct type of structure found at the address provided in *Object*. The command assumes that this is the type of structure found at *Object* and displays data accordingly.

A listing of all supported objects can be retrieved by issuing a [!ks.forcedump](#) command with no arguments.

Here are two examples of the output from [!ks.forcedump](#), using the address of a filter for the *Object* argument but with different levels of detail:

```
dbgcmd

kd> !forcedump 829493c4 KSFILTER
WARNING: I am dumping 829493c4 as a KSFILTER.
          No checking has been performed to ensure that it is this type!!!

Filter object 829493c4 [CKsFilter = 82949350]
  Descriptor      f7a233c8:
  Context        829dce28

kd> !forcedump 829493c4 KSFILTER 7
WARNING: I am dumping 829493c4 as a KSFILTER.
          No checking has been performed to ensure that it is this type!!!

Filter object 829493c4 [CKsFilter = 82949350]
  Descriptor      f7a233c8:
  Filter Category GUIDs:
    Video
    Capture
  Context        829dce28
  INTERNAL INFORMATION:
    Public Parent Factory   829782dc
    Aggregated Unknown     00000000
```

```
Device Interface      823b83c8
Control Mutex        829493f8 is not held
Object Event List:
    None
CKsFilter object 82949350 [KSFILTER = 829493c4]
    Processing Mutex      82949484 is not held
    Gate &                82949464
    Gate.Count            1
Pin Factories:
    Pin ID 0 [Video/General Capture Out]:
        Child Count       1
        Bound Child Count 1
        Necessary Count   0
    Specific Instances:
        8293f580
```

!ks.graph

Article • 10/25/2023

The **!ks.graph** extension command displays a textual description of the kernel mode graph in topologically sorted order.

dbgcmd

```
!ks.graph Object [Level] [Flags]
```

Parameters

Object

Specifies a pointer to the object to use as a starting point for the graph. Must be a pointer to one of the following: file object, IRP, pin, or filter.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7. The levels for **!ks.graph** are the same as those for [!ks.dump](#).

Flags

Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)

Display a list of IRPs queued to each pin instance in the graph.

Bit 1 (0x2)

Display a list of IRPs that are pending from each pin instance in the graph. Only IRPs that the pin knows it is waiting for are displayed.

Bit 4 (0x10)

Analyze a stalled graph for suspect filters.

DLL

Windows 2000

winxp\Ks.dll

Windows XP and later

Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

This command may take a bit of time to process.

Issue a **!ks.graph** command with no arguments for help.

Here is an example of the **!ks.graph** display, with the address of a filter object:

```
dbgcmd

kd> !graph 829493c4
Attempting a graph build on 829493c4... Please be patient...

Graph With Starting Point 829493c4:
"avssamp" Filter 82949350, Child Factories 1
    Output Factory 0 [Video/General Capture]:
        Pin 8293f4f0 (File 82503498) Irps(q/p) = 2, 0
```

!ks.libexts

Article • 10/25/2023

The **!ks.libexts** extension provides access to Microsoft-supplied library extensions that are statically linked to the extension module.

dbgcmd

```
!ks.libexts [Command] [Libext]
```

Parameters

Command

Optional. Specifies one of the following values. If this argument is omitted, **!ks.libexts** returns help information.

disableall

Disable all library extensions. When this is used, omit the *Libext* parameter.

disable

Disable a specific library extension by name. When this is used, specify the name in the *Libext* parameter.

enableall

Enable all library extensions. Only loaded components with correct symbols are enabled. When this is used, omit the *Libext* parameter.

enable

Enable a specific library extension by name. When this is used, specify the name in the *Libext* parameter. Only loaded components with correct symbols can be enabled.

details

Show details about all currently linked library extensions. When this is used, omit the *Libext* parameter.

Libext

Specifies the name of a library extension. Required only for *Command* values of **enable** or **disable**.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The extension module contains an extensibility framework that allows separate components to be built and linked into Ks.dll. These extra components are called library extensions.

The **!ks.libexts** command allows viewing of statistics about those library extensions as well as control over them. For details, issue **!ks.libexts** with no arguments.

!ks.objhdr

Article • 10/25/2023

The **!ks.objhdr** extension displays the kernel streaming object header associated with the specified file object.

dbgcmd

```
!ks.objhdr FileObject [Level] [Flags]
```

Parameters

FileObject

This parameter specifies a pointer to a WDM file object. If *FileObject* is not valid, the command returns an error.

Level

Optional. Values are the same as those for [!ks.dump](#).

Flags

Optional. Values are the same as those for [!ks.dump](#).

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Levels and flags for **!ks.objhdr** are identical to those described in [!ks.dump](#).

The output from [!ks.allstreams](#) and [!ks.enumdevobj](#) can be used as the input for **!ks.objhdr**. To do this with the *avssamp* sample, for instance, issue the following commands:

```
dbgcmd
```

```
kd> !ks.allstreams
6 Kernel Streaming FD0s found:
    Functional Device 8299be18 [\Driver\smwdm]
    Functional Device 827c86d8 [\Driver\wdmaud]
    Functional Device 827c0f08 [\Driver\sysaudio]
    Functional Device 82424590 [\Driver\avssamp]
    Functional Device 82423720 [\Driver\MSPQM]
    Functional Device 82b91a88 [\Driver\MSPCLOCK]
kd> !ks.enumdevobj 82424590
WDM device object 82424590:
    Corresponding KSDEVICE      82427540
    Factory 8285baa4 [Descriptor f7a333c8] instances:
        82837a34
kd> !ks.objhdr 82837a34 7
```

The results of this command might be lengthy. Issue a Ctrl-BREAK (WinDbg) or Ctrl-C (NTSD, CDB, KD) to stop the output.

Here's a separate example:

```
dbgcmd
```

```
kd> !ks.objhdr 81D828B8 7
Adjusting file object 81D828B8 to object header 81BC1008

Object Header 81BC1008
    Associated Create Items:
        Create Item F9F77E98
            CreateFunction = ks!CKsFilter::DispatchCreatePin+0x00
            ObjectClass = {146F1A80-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
    Child Create Handler List:
        Create Item F9F85AA0
            CreateFunction = ks!CKsPin::DispatchCreateAllocator+0x00
            ObjectClass = {642F5D00-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
        Create Item F9F85AB8
            CreateFunction = ks!CKsPin::DispatchCreateClock+0x00
            ObjectClass = {53172480-4791-11D0-A5D6-28DB04C10000}
            Flags = 0
        Create Item F9F85AD0
            CreateFunction = ks!CKsPin::DispatchCreateNode+0x00
            ObjectClass = {0621061A-EE75-11D0-B915-00A0C9223196}
            Flags = 0
    DispatchTable:
        Dispatch Table F9F85AE8
            DeviceIoControl = ks!CKsPin::DispatchDeviceIoControl+0x00
            Read = ks!KsDispatchInvalidDeviceRequest+0x00
            Write = ks!KsDispatchInvalidDeviceRequest+0x00
            Flush = ks!KsDispatchInvalidDeviceRequest+0x00
```

```
Close = ks!CKsPin::DispatchClose+0x00
QuerySecurity = ks!KsDispatchQuerySecurity+0x00
SetSecurity = ks!KsDispatchSetSecurity+0x00
FastDeviceIoControl =
ks!KsDispatchFastIoDeviceControlFailure+0x00
    FastRead = ks!KsDispatchFastReadFailure+0x00
    FastWrite = ks!KsDispatchFastReadFailure+0x00
TargetState: KSTARGET_STATE_ENABLED
TargetDevice: 00000000
BaseDevice : 81BBDF10
Stack Depth : 1
Corresponding AVStream object = 81A971B0
```

!ks.ohdr

Article • 10/25/2023

The **!ks.ohdr** extension displays details of a kernel streaming object header.

dbgcmd

```
!ks.ohdr Object [Level] [Flags]
```

Parameters

Object

This parameter specifies a pointer to a KS object header. If *Object* is not valid, the command returns an error.

Level

Optional. Values are the same as those for [!ks.dump](#).

Flags

Optional. Values are the same as those for [!ks.dump](#).

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

The **!ks.ohdr** command works similarly to [!ks.objhdr](#) in that it displays details of a KS object header. The difference is that the caller provides the direct address of the KS object header, instead of the address of the associated file object.

Levels and flags for **!ks.ohdr** are identical to those described in [!ks.dump](#).

If the data you are querying is not paged out, consider using [**!ks.dump**](#) instead of [**!ks.ohdr**](#).

!ks.pciaudio

Article • 10/25/2023

The **!ks.pciaudio** extension displays a list of FDOs currently attached to PortClIs.

dbgcmd

!ks.pciaudio [Options] [Level]

Parameters

Options Optional. Specifies the kind of information to be displayed. *Options* can be any combination of the following bits.

Bit 0 (0x1)

Display a list of running streams.

Bit 1 (0x2)

Display a list all streams.

Bit 3 (0x4)

Output displayed streams. *Level* has meaning only when this bit is set.

Level

Optional, and applicable only if Bit 3 is set in *Options*. Levels are the same as those for [!ks.dump](#).

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

Here is an example of the output from **!ks.pciaudio**:

dbgcmd

```
kd> !ks.pciaudio
1 Audio FDOs found:
Functional Device 8299be18 [\Driver\smwdm]
```

!ks.pciks

Article • 10/25/2023

The **!ks.pciks** extension lists functional devices for kernel streaming devices that are attached to the PCI bus. Optionally, it can display information about active streams on those functional devices.

dbgcmd

```
!ks.pciks [Flags] [Level]
```

Parameters

Flags

Optional. Specifies the kind of information to be displayed. *Flags* can be any combination of the following bits.

Bit 0 (0x1)

List all currently running streams.

Bit 1 (0x2)

Recurse graphs to find non-PCI devices.

Bit 2 (0x4)

Display a list of proxy instances.

Bit 3 (0x8)

Display currently queued Irps.

Bit 4 (0x10)

Display information about all streams.

Bit 5 (0x20)

Display active stream formats.

Level

Optional, and applicable only to flag combinations that cause data to be displayed.

Levels are the same as those for [!ks.dump](#).

DLL

Windows 2000

winxp\Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

This command may take time to execute, especially if the ACPI filter driver is loaded, or if Driver Verifier is enabled and driver names are paged out.

Here is an example of the !ks.pciks display:

```
dbgcmd  
  
kd> !pciks  
1 Kernel Streaming FDOs found:  
    Functional Device 82a17690 [\Driver\smwdm]
```

!ks.topology

Article • 10/25/2023

The **!ks.topology** extension displays a sorted graph of the internal topology of the filter closest to *Object*.

dbgcmd

```
!ks.topology Object [Level] [Flags]
```

Parameters

Object

Specifies a pointer to the object to use as a base for the graph. Can be a pointer to a file object, IRP, pin, filter, or other KS object.

Level

Optional. Specifies the level of detail to display on a 0-7 scale with progressively more information displayed for higher values. To display all available details, supply a value of 7.

Flags

Not currently available.

DLL

Windows 2000	winxp\Ks.dll
Windows XP and later	Ks.dll

Additional Information

For more information, see [Kernel Streaming Debugging](#).

Remarks

For help, issue a **!ks.topology** command with no arguments.

Note that this command may take a few moments to execute.

!acxkd

Article • 06/21/2024

The **!acxkd** extension displays information about audio class extension (ACX) drivers. For more information about ACX, see [ACX audio class extensions overview](#).

Syntax

dbgcmd

!acxkd. [Options]

DLL

Acxkd.dll

Debugger Version

The **!acxkd** extension is available in WinDbg version 1.2402.24001.0 and later.

ACX 1.0 debugging

The **!acxkd** debugger extension offers only partial functionality under ACX 1.0. Updating to ACX 1.1 is recommended.

Parameters

Options - Specifies the type of information to be display.

[] [Expand table](#)

Option	Description	Parameter
!help	Displays information on available extension commands.	<code>[<command name>]</code>
!acxcircuit	Dump an ACXCIRCUIT object.	<code><circuit></code> - ACXCIRCUIT WDF handle
!acxdataformat	Dump an ACXDATAFORMAT object.	<code><dataformat></code> - ACXDATAFORMAT

Option	Description	Parameter
		WDF handle
!acxdataformatlist	Dump an ACXDATAFORMATLIST object.	<dataformatlist> - ACXDATAFORMATLIST WDF handle
!acxdevice	Dump an ACXDEVICE object.	<device> - ACXDEVICE WDF handle
!acxelement	Dump an ACXELEMENT object.	<element> - ACXELEMENT WDF handle
!acxevents	Dump events of an ACXObject object.	<events> - ACXObject WDF handle
!acxfactory	Dump an ACXFACTORYCIRCUIT object.	<factory> - ACXFACTORYCIRCUIT WDF handle
!acxmanager	Dump an ACXMANAGER object.	None
!acxmethods	Dump methods of an ACXObject object.	<object> - ACXObject WDF handle
!acxobjbag	Dump an ACXObjectBAG object.	<objbag> - ACXObjectBAG WDF handle
!acxobject	Dump an ACXObject object.	<object> - ACXObject WDF handle
!acxpin	Dump an ACXPIN object.	<pin> - ACXPIN WDF handle
!acxproperties	Dump properties of an ACXObject object.	<properties> - ACXObject WDF handle
!acxstream	Dump an ACXSTREAM object.	<stream> - ACXSTREAM handle
!acxstreambridge	Dump an ACXSTREAMBRIDGE object.	<bridge> - ACXSTREAMBRIDGE handle
!acxtarget	Dump an ACXTARGET object.	<target> - ACXTARGET WDF handle
!acxtemplate	Dump an ACXTEMPLATE object.	<ttmp> - ACXTEMPLATE handle

Remarks

!acxkd.help

To list all available commands, use the acxkd `!help` command.

```
0: kd> !acxkd.help
Commands for C:\Debugger\acxkd.dll:
!acxcircuit          - Dump a ACXCIRCUIT object.
!acxdataformat       - Dump a ACXDATAFORMAT object.
!acxdataformatlist   - Dump a ACXDATAFORMATLIST object.
!acxdevice           - Dump a ACXDEVICE object.
!acxelement          - Dump a ACXELEMENT object.
!acxevents           - Dump events of a ACXObject object.
!acxfactory          - Dump a ACXFACTORYCIRCUT object.
!acxmanager          - Dump a ACXMANAGER object.
!acxmethods          - Dump methods of a ACXObject object.
!acxobjbag           - Dump a ACXObjectBAG object.
!acxobject           - Dump a ACXObject* object.
!acxpin              - Dump a ACXPIN object.
!acxproperties        - Dump properties of a ACXObject object.
!acxstream            - Dump a ACXSTREAM object.
!acxstreambridge     - Dump a ACXSTREAMBRIDGE object.
!acxtarget            - Dump a ACXTARGET* object.
!acxtemplate          - Dump a ACX*TEMPLATE object.
!help                - Displays information on available extension commands
!help <cmd> will give more information for a particular command
```

Use the acxkd `!help` command to learn more about any of the commands, for example the `!acxdevice` command.

```
dbgcmd

0: kd> !acxkd.help acxdevice
!acxdevice <device>
<device> - ACXDEVICE handle
Dump a ACXDEVICE object.
```

Use the `!acxdevice` command as a starting point to examine the ACX driver.

```
dbgcmd

3: kd> !acxdevice 0x00007dfadb0a5358
Dumping info for ACXDEVICE 0x00007dfadb0a5358

In connected standby: FALSE

State: AfxDeviceStateInitialized
State history:
 0 : AfxDeviceStateInvalid
 1 : AfxDeviceStateInvalid
 2 : AfxDeviceStateInvalid
 3 : AfxDeviceStateInvalid
 4 : AfxDeviceStateInvalid
 5 : AfxDeviceStateCreated
 6 : AfxDeviceStateInitializing
```

```

7 : AfxDeviceStateInitialized

Create dispatch list:
    Create name: eHDMIOutTopo
    Dispatch routine: fffff80393179918
    Dispatch context: ffff82052513b960

Circuits:
-----
[Circuit 0]

Name: eHDMIOutTopo
Type: AcxCircuitTypeRender
ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}

State: AfxCircuitStatePoweredUp

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49
48348&0&0002&00000025#{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo

!acxproperties 00007dfad9cccb8
!acxmethode 00007dfad9cccb8
!acxevents 00007dfad9cccb8

# Pins: 2
    !acxpin 00007dfadf996dd8
    !acxpin 00007dfad4697238

# Elements: 1
    !acxelement 00007dfadf997a18

# Streams: 0

!acxcircuit 00007dfad9cccb8

!wdfqueue 00007dfade9beaf8
!wdfdevice 00007dfadb0a5358

!wdfhandle 00007dfadb0a5358
dt Acx01000!Acx::AfxDevice ffff8205256ab420

```

Click on the links in the output to display information using the !acxproperties, !acxmethode and !acxevents commands.

For information on locating the wdfhandle for the ACXDEVICE object, see the [Example ACX driver walkthrough](#) in this article.

WDF commands - !wdfkd.wdfldr

As ACX drivers are WDF drivers, use any of the WDF kernel debugger commands. For example, use `!wdfkd.wdfldr` to display version information and the ACX binding with WDF.

```
dbgcmd

0: kd> !wdfldr acx01000
WDF Driver: Acx01000
-----
CLIENT_MODULE 0xfffff82052abdc0
    WDF Version v1.31
    ImageName   Acx01000.sys
    ImageAddress 0xfffff80393150000
    ImageSize    0xb3000
    BindingList  0xfffff82052abdcc08

    ImageName      WdfVer Ver   WdfGlobals           BindInfo
    ImageAddress   ImageSize
    Wdf01000.sys   v1.33  v1.33  0xfffff8205218d8fb0 0xfffff8205218d8df0
0xfffff80356a00000 0x000c7000
-----
CLASS_MODULE 0xfffff820525b3dc90
    WDF Version v1.31
    Version     v1.1
    Service     \Registry\Machine\System\CurrentControlSet\Services\acx01000
    ImageName   Acx01000.sys
    ImageAddress 0xfffff80393150000
    ImageSize    0xb3000
    ClientsList  0xfffff820525b3dcf8
    Associated Clients: 1

    ImageName      WdfVer Ver   WdfGlobals           BindInfo
    ImageAddress   ImageSize
    AcxHdAudio.sys v1.25  v1.0   0xfffff820527f70ae0 0xfffff803930df3e8
0xfffff803930c0000 0x0008a000
-----
```

!acxkd.acxmanager

Use the `!acxmanager` command to display information about the ACXMANAGER object. This provides a good starting point to investigate ACX drivers.

This example shows the first part of the extensive `!acxmanager` output provided for a multircuit ACX configuration.

```
dbgcmd

10: kd> !acxmanager
Dumping info for ACXMANAGER 0x000054f94d1d4378
```

```
Delete pending: No

# singleton composites: 8
-----
[Composite 0]

State: AfxCompositeStateActive
!acxobjbag 000054f94c8e61c8
!acxtemplate 000054f94c014d28
!acxobject 000054f94c0141c8

# circuits: 3
-----
[Circuit 0 CORE]
...
```

This example output shows a single ACX circuit.

```
dbgcmd

0: kd> !acxmanager
Dumping info for ACXMANAGER 0x000049f6c3c769f8

Delete pending: No

# singleton composites: 0
# Composite factories: 0
!wdfhandle 000049f6c3c769f8
dt Acx01000!Acx::AfxManager fffffb6093c3896b0
```

!acxkd.acxobject

In the output for the !acxmanager an address is provided for the wdfhandle. Use the wdfhandle address with the `!acxobject` command to display information about the ACXMANAGER or any other ACX object.

```
dbgcmd

0: kd> !acxobject 000049f6c3c769f8
Dumping info for ACXMANAGER 0x000049f6c3c769f8

Delete pending: No

# singleton composites: 0
# Composite factories: 0
!wdfhandle 000049f6c3c769f8
dt Acx01000!Acx::AfxManager fffffb6093c3896b0
```

Click on the `dt` link in the output shown above, to see more information about the internal ACX object structures.

```
0: kd> dt Acx01000!Acx::AfxManager fffffb6093c3896b0
+0x000 m_Object          : 0x000049f6`c3c769f8 ACXMANAGER_
=fffffb803`8a478ad8 s_AfxManager      : 0xfffffb609`3c3896b0 Acx::AfxManager
+0x008 m_AcxGlobals       : 0xfffffb609`403cae24 _ACX_DRIVER_GLOBALS
+0x010 m_Flags           : 0
+0x010 m_Unloading       : 0y0
+0x018 m_CompositesMutex : _FAST_MUTEX
+0x050 m_CompositeFactoriesList : _LIST_ENTRY [ 0xfffffb609`3c389700 -
0xfffffb609`3c389700 ]
+0x060 m_CompositeFactoriesCount : 0n0
+0x068 m_CompositesList   : _LIST_ENTRY [ 0xfffffb609`3c389718 -
0xfffffb609`3c389718 ]
+0x078 m_CompositesCount : 0n0
+0x080 m_FactoriesList    : _LIST_ENTRY [ 0xfffffb609`3c389730 -
0xfffffb609`3c389730 ]
+0x090 m_CircuitsList     : _LIST_ENTRY [ 0xfffffb609`40a27068 -
0xfffffb609`40a27068 ]
+0x0a0 m_FiltersMutex     : _FAST_MUTEX
+0x0d8 m_FactoryFiltersList : _LIST_ENTRY [ 0xfffffb609`3c389788 -
0xfffffb609`3c389788 ]
+0x0e8 m_CircuitFiltersList : _LIST_ENTRY [ 0xfffffb609`3c389798 -
0xfffffb609`3c389798 ]
+0x0f8 m_FactoriesNotificationHandle : 0xfffffc70c`b2356790 Void
+0x100 m_CircuitsNotificationHandle : 0xfffffc70c`b2357ec0 Void
+0x108 m_CommandQueue     : 0xfffffb609`3ffc5400 Acx::AfxWorkQueue
```

Note that the debugger output may contain a mix of public ACX objects such as ACXMANAGER, ACXCIRCUITFACTORY, and ACXCIRCUIT and internal structures that are defined to be opaque. The internal types are not guaranteed to stay the same, or be available in different releases of ACX, and must not be called or used directly.

Since ACX objects are WDF objects, you can use the `!wdfkd.wdfhandle` command to display additional information about the ACXMANAGER object.

```
0: kd> !wdfhandle 000049f6c3c769f8
Treating handle as a KMDF handle!

Dumping WDFHANDLE 0x000049f6c3c769f8
=====
Handle type is WDOBJECT [ACXMANAGER]
RefCount: 2
Contexts:
    context: dt 0xfffffb6093c3896b0 Acx01000!AfxManager (size is 0x110)
```

```
bytes)
    EvtCleanupCallback ffffff8038a453070
Acx01000!Acx::WdfCpp::ObjectContext<ACXMANAGER__>
* ,Acx::AfxManager>::EvtObjectContextCleanupThunk
    EvtDestroyCallback ffffff8038a453000
Acx01000!Acx::WdfCpp::ObjectContext<ACXMANAGER__>
* ,Acx::AfxManager>::EvtObjectContextDestroyThunk

    context: dt 0xfffffb609404c2280 Acx01000!WdfCustomType_ACXMANAGER (size
is 0x10 bytes)
    <no associated attribute callbacks>

Parent: !wdfhandle 0x000049f6d4e2d5c8, type is WDFDEVICE
Owning device: !wdfdevice 0x000049f6d4e2d5c8

!wdfobject 0xfffffb6093c389600
```

!acxkd.acxpin

Commands that display other ACX information, such as `!acxpin` require a WDF handle to the object. For information on locating the WDF handle for an ACX object, see [Example ACX driver walkthrough](#) in this article.

```
dbgcmd

0: kd> !acxpin 0x000049f6befeee38
Dumping info for ACXPIN 0x000049f6befeee38

ID: 0
Type: AcxPinTypeSink
Type: AcxPinCommunicationSink

Category: KSCATEGORY_AUDIO
Name: {00000000-0000-0000-0000-000000000000}
```

Depending on the state of the ACX object, not all information may be available to display.

!acxkd.acxdataformatlist

Similar to `!acxpin`, `!acxdataformatlist` displays information on ACX dataformat lists.

```
dbgcmd

0: kd> !acxdataformatlist 0x000049f6bf8be668
Dumping info for ACXDATAFORMATLIST 0x000049f6bf8be668

# Scan count: 0
```

```
# Data formats: 6
Data formats:
    Sample Rate: 48000, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3 (default)
        Sample Rate: 48000, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3
        Sample Rate: 44100, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3
        Sample Rate: 44100, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3
        Sample Rate: 32000, #Channels: 2, #Bits: 16, ValidBits: 16, Mask: 0x3
        Sample Rate: 32000, #Channels: 2, #Bits: 32, ValidBits: 24, Mask: 0x3

!wdfhandle 000049f6bf8be668
dt Acx01000!Acx::AfxDataFormatList fffffb60940444530
```

Example ACX driver walkthrough

This section provides a walkthrough of debugging an ACX driver.

Symbols path

Use [.symfix](#) and the [.sympath \(Set Symbol Path\)](#) commands to change the symbol path. If you're using local code with the driver add the path to that code as well. Use the [.reload \(Reload Module\)](#) command to reload symbols from the current path.

```
dbgcmd

.symfix
.sympath+ C:\Windows-driver-samples-
develop\audio\Acx\Samples\AudioCodec\Driver
.reload /f
```

Debugger context for drivers

If you're debugging an active ACX driver, set a breakpoint. This places the debugger in the context of the ACX objects, to be able to gather and display information.

These example breakpoints are designed to fire as the sample AudioCodec driver starts up.

```
dbgcmd
```

```
bm AudioCodec!DriverEntry  
bm AudioCodec!AcxDriverInitialize
```

These example breakpoints are designed to fire when specific actions occur, such as pin or circuit creation.

```
dbgcmd  
  
bm AudioCodec!AcxPinCreate  
bm AudioCodec!AcxCircuitCreate  
bm AudioCodec!Codec_EvtBusDeviceAdd
```

Once the driver is loaded and the appropriate breakpoint has fired, and valid execution context is available, use !acxkd commands to display information about any ACX object. Use the !acxobject command for general information and a specific command, such as !acxcircuit or !acxpin for more granular information.

Load the acxkd dll

Use the [.load \(Load Extension DLL\)](#) command to load the acxkd.dll extension.

```
dbgcmd  
  
.load acxkd.dll
```

Display information about the ACX driver

To gather information about the target driver, use the [!m \(List Loaded Modules\)](#) command to see all loaded drivers. Then use the Dvm options to display information about the ACX driver of interest as shown here.

```
dbgcmd  
  
0: kd> !m Dvm AcxHdAudio  
Browse full module list  
start end module name  
fffff803`8a3c0000 fffff803`8a448000 AcxHdAudio (private pdb symbols)  
C:\ProgramData\Debug\sym\AcxHdAudio.pdb\6AEA2622909B20C1AD149C57ACBB4A6F1\AcxH  
dAudio.pdb  
    Loaded symbol image file: AcxHdAudio.sys  
    Mapped memory image file:  
C:\ProgramData\Debug\sym\AcxHdAudio.sys\082942338800\AcxHdAudio.sys  
        Image path: \SystemRoot\System32\drivers\AcxHdAudio.sys  
        Image name: AcxHdAudio.sys
```

```
Browse all global symbols  functions  data  Symbol Reload
Image was built with /Bprepro flag.
Timestamp:          08294233 (This is a reproducible build file hash, not
a timestamp)
CheckSum:           00087DD6
ImageSize:          00088000
Translations:       0000.04b0 0000.04e4 0409.04b0 0409.04e4
Information from resource tables:
```

Use the [x \(Examine Symbols\)](#) command and a wildcard mask to display specific ACX structures, such as an ACXPIN.

```
dbgcmd
```

```
0: kd> x /D AcxHdAudio!acxpin*
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

fffff803`8a3e3216 AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+66
fffff803`8a3e31e0 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+30
fffff803`8a3e361f AcxHdAudio!AcxPinNotifyDataFormatChange = (inline caller)
AcxHdAudio!HDACodec_EvtFormatChange+46f
fffff803`8a3e7396 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+962
fffff803`8a3e74a2 AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+a6e
fffff803`8a3e75f8 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+bc4
fffff803`8a3e789a AcxHdAudio!AcxPinAddJacks = (inline caller)
AcxHdAudio!HDACodecR_CreateRenderCircuit+e66
fffff803`8a3e5ae5 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodecR_EvtAcxPinRetrieveJackSinkInfo+25
fffff803`8a3e5913 AcxHdAudio!AcxPinGetCircuit = (inline caller)
AcxHdAudio!HDACodecR_EvtAcxPinRetrieveName+73
fffff803`8a3ea277 AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+b27
fffff803`8a3ea39c AcxHdAudio!AcxPinGetRawDataFormatList = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+c4c
fffff803`8a3ea4bc AcxHdAudio!AcxPinCreate = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+d6c
fffff803`8a3ea7b2 AcxHdAudio!AcxPinAddJacks = (inline caller)
AcxHdAudio!HDACodecC_CreateCaptureCircuit+1062
```

Depending on the current execution context in the debugger, it may be possible to use the [dx \(Display Debugger Object Model Expression\)](#) to drill down into specific ACX structures.

```
dbgcmd
```

```

0: kd> dx -r2 AudioCodec!AcxDeviceAddCircuit
AudioCodec!AcxDeviceAddCircuit          :
AudioCodec!AcxDeviceAddCircuit+0x0 [Type: long __cdecl(WDFDEVICE_
*,ACXCIRCUIT_ *)]
0: kd> u ffffff8007ead1120
AudioCodec!AcxDeviceAddCircuit [C:\Program Files (x86)\Windows
Kits\10\Include\10.0.26016.0\km\acx\km\1.1\AcxDevice.h @ 206]:
fffff800`7ead1120 4889542410      mov     qword ptr [rsp+10h],rdx
fffff800`7ead1125 48894c2408      mov     qword ptr [rsp+8],rcx
fffff800`7ead112a 4883ec38       sub    rsp,38h
fffff800`7ead112e b808000000      mov     eax,8
fffff800`7ead1133 486bc046       imul   rax,rax,46h
fffff800`7ead1137 488d0dc2ab0000  lea    rcx,[AudioCodec!AcxFunctions
(fffff800`7eadbd00)]
fffff800`7ead113e 488b0401       mov     rax,qword ptr [rcx+rax]
fffff800`7ead1142 4889442420      mov     qword ptr [rsp+20h],rax

```

!wdfkd.wdfdriverinfo

Use the [!wdfdriverinfo](#) command with the name of the driver to gather WDF information, such as the associated ACX objects and ACXDEVICE wdfhandle.

```

dbgcmd

0: kd> !wdfdriverinfo AcxHdAudio.sys 1 -v
-----
Default driver image name: AcxHdAudio
WDF library image name: Wdf01000
  FxDriverGlobals 0xfffffb609403e3de0
  WdfBindInfo     0xfffff8038a3dd1b0
    Version        v1.25
  Library module  0xfffffb60929cc6050
    ServiceName
\Registry\Machine\System\CurrentControlSet\Services\Wdf01000
  ImageName      Wdf01000
-----
WDFDRIVER: 0x000049f6bfe9d5d8
  context: dt 0xfffffb60940162bc0 AcxHdAudio!CODEC_DRIVER_CONTEXT (size is
0x1 bytes)
    <no associated attribute callbacks>

  context: dt 0xfffffb6093ccead20 Acx01000!AfxDriver (size is 0x20 bytes)
    EvtCleanupCallback ffffff8038a455780
Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_>
*,Acx::AfxDriver>::EvtObjectContextCleanupThunk
    EvtDestroyCallback ffffff8038a455740
Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_>
*,Acx::AfxDriver>::EvtObjectContextDestroyThunk
Object Hierarchy: !wdfhandle 0x000049f6bfe9d5d8 0xff
Driver logs: !wdflogdump AcxHdAudio.sys -d
Framework logs: !wdflogdump AcxHdAudio.sys -f

```

```

!wdfdevice 0x000049f6bffba488 ff (FDO)
    Pnp/Power State: WdfDevStatePnpStarted, WdfDevStatePowerD0,
    WdfDevStatePwrPolStartedWakeCapable
        context: dt 0xfffffb60940045e60 AcxHdAudio!CODEC_DEVICE_CONTEXT
    (size is 0x110 bytes)
        EvtCleanupCallback fffff8038a3e3c90
    AcxHdAudio!HDACodec_EvtDeviceContextCleanup

        context: dt 0xfffffb60933f030f0 Acx01000!AfxDevice (size is 0x150
    bytes)
        EvtCleanupCallback fffff8038a451910
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextCleanupThunk
        EvtDestroyCallback fffff8038a451740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextDestroyThunk

        context: dt 0xfffffb60940a26b90
    AcxHdAudio!CODEC_RENDER_DEVICE_CONTEXT (size is 0x38 bytes)
        <no associated attribute callbacks>

        context: dt 0xfffffb609409f2e00
    AcxHdAudio!CODEC_CAPTURE_DEVICE_CONTEXT (size is 0x8 bytes)
        <no associated attribute callbacks>
    !wdfdevicequeues 0x000049f6bffba488

    !wdfdevice 0x000049f6bef1a848 ff (PDO)
        Pnp/Power State: WdfDevStatePnpStarted,
    WdfDevStatePowerD0BusWakeOwner, WdfDevStatePwrPolStartedWakeCapable
        context: dt 0xfffffb609410e5aa0
    AcxHdAudio!CODEC_RENDER_DEVICE_CONTEXT (size is 0x38 bytes)
        EvtCleanupCallback fffff8038a3e6400
    AcxHdAudio!HDACodecR_EvtDeviceContextCleanup

        context: dt 0xfffffb60933f090d0 Acx01000!AfxDevice (size is
    0x150 bytes)
        EvtCleanupCallback fffff8038a451910
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextCleanupThunk
        EvtDestroyCallback fffff8038a451740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDEVICE_>
    *,Acx::AfxDevice>::EvtObjectContextDestroyThunk
    !wdfdevicequeues 0x000049f6bef1a848

-----
WDF Verifier settings for AcxHdAudio.sys is OFF
-----

```

In the output shown above, a link is available to the associated wdfdevice. Click on that link to display information about the associated WDF device objects.

```
dbgcmd
```

```
0: kd> !wdfdevice 0x000049f6bef1a848 ff
Treating handle as a KMDF handle!

Dumping WDFDEVICE 0x000049f6bef1a848
=====
WDM PDEVICE_OBJECTs: self fffffb60940ddbdd0

Pnp state: 119 ( WdfDevStatePnpStarted )
Power state: 309 ( WdfDevStatePowerD0BusWakeOwner )
Power Pol state: 531 ( WdfDevStatePwrPolStartedWakeCapable )
```

!acxkd.acxdevice

Using the same wdfhandle with `!acxdevice` provides ACX centric information.

```
dbgcmd
```

```
3: kd> !acxdevice 0x00007dfadb0a5358
Dumping info for ACXDEVICE 0x00007dfadb0a5358
```

```
In connected standby: FALSE
```

```
State: AfxDeviceStateInitialized
```

```
State history:
```

```
0 : AfxDeviceStateInvalid
1 : AfxDeviceStateInvalid
2 : AfxDeviceStateInvalid
3 : AfxDeviceStateInvalid
4 : AfxDeviceStateInvalid
5 : AfxDeviceStateCreated
6 : AfxDeviceStateInitializing
7 : AfxDeviceStateInitialized
```

```
Create dispatch list:
```

```
Create name: eHDMIOutTopo
```

```
Dispatch routine: fffff80393179918
```

```
Dispatch context: ffff82052513b960
```

```
Circuits:
```

```
-----
```

```
[Circuit 0]
```

```
Name: eHDMIOutTopo
```

```
Type: AcxCircuitTypeRender
```

```
ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}
```

```
State: AfxCircuitStatePoweredUp
```

```

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49
48348&0&0002&00000025#\{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo

!acxproperties 00007dfad9ccccb8
!acxmethods 00007dfad9ccccb8
!acxevents 00007dfad9ccccb8

# Pins: 2
    !acxpin 00007dfadf996dd8
    !acxpin 00007dfad4697238

# Elements: 1
    !acxelement 00007dfadf997a18

# Streams: 0

!acxcircuit 00007dfad9ccccb8

!wdfqueue 00007dfade9beaf8
!wdfdevice 00007dfadb0a5358

!wdfhandle 00007dfadb0a5358
dt Acx01000!Acx::AfxDevice ffff8205256ab420

```

To display information about the other ACX objects, such as the ACXCIRCUIT, use the link in output above that invokes `!acxcircuit` with the appropriate wdfhandle.

dbgcmd

```

3: kd> !acxcircuit 00007dfad9ccccb8
Dumping info for ACXCIRCUIT 0x00007dfad9ccccb8

Name: eHDMIOutTopo
Type: AcxCircuitTypeRender
ComponentId: {BFCA9AD9-4EED-46C2-9323-B5D4400761A5}

State: AfxCircuitStatePoweredUp
State history:
    0 : AfxCircuitStateInvalid
    1 : AfxCircuitStateInvalid
    2 : AfxCircuitStateInvalid
    3 : AfxCircuitStateCreated
    4 : AfxCircuitStateInitializing
    5 : AfxCircuitStateInitialized
    6 : AfxCircuitStatePoweredDown
    7 : AfxCircuitStatePoweredUp

Interface is enabled
SymbolicLinkName: \??
\HDAUDIO#SUBFUNC_01&VEN_8086&DEV_281F&NID_0001&SUBSYS_00000000&REV_1000#6&49

```

```
48348&0&0002&00000025#{2c6bb644-e1ae-47f8-9a2b-1d1fa750f2fa}\eHDMIOutTopo
```

```
# Power references: 0
# Open handles: 18

!acxproperties 00007dfad9ccccb8
!acxmethods 00007dfad9ccccb8
!acxevents 00007dfad9ccccb8

# Pins: 2
!acxpin 00007dfadf996dd8
!acxpin 00007dfad4697238

# Elements: 1
!acxelement 00007dfadf997a18
```

!acxkd.acxproperties

In the output command links with the wdfhandle are provided for other objects such as ACX properties, that can be displayed.

```
dbgcmd
```

```
0: kd> !acxproperties 000049f6bf436b88
Dumping properties info for ACXObject 0x000049f6bf436b88

# sets: 4

Set: {8C134960-51AD-11CF-878A-94F801C10000}
  Id: 0, Flags: 0x1
  Id: 1, Flags: 0x1

Set: {720D4AC0-7533-11D0-A5D6-28DB04C10000}
  Id: 0, Flags: 0x1
  Id: 1, Flags: 0x1
  Id: 2, Flags: 0x1
  Id: 3, Flags: 0x1

Set: {C034FDB0-FF75-47C8-AA3C-EE46716B50C6}
  Id: 1, Flags: 0x1
  Id: 2, Flags: 0x1
  Id: 3, Flags: 0x1

Set: {4D12807E-55DB-48B8-A466-F15A510F5817}
  Id: 1, Flags: 0x1
```

!wdfhandle

Also available in the `!wdfdriverinfo` output is a link to a wdfhandle for the WDF object hierarchy associated with ACX.

```
dbgcmd
```

```
Object Hierarchy: !wdfhandle 0x000049f6bfe9d5d8 0xff
```

Clicking on that link displays the WDF object hierarchy for the ACX driver. This output can be used to locate WDF handles for other ACX and WDF objects.

```
dbgcmd
```

```
0: kd> !wdfhandle 0x000049f6bfe9d5d8 0xff
Treating handle as a KMDF handle!

Dumping WDFHANDLE 0x000049f6bfe9d5d8
=====
Handle type is WDFDRIVER
RefCount: 1
Contexts:
    context: dt 0xfffffb60940162bc0 AcxHdAudio!CODEC_DRIVER_CONTEXT (size is
0x1 bytes)
        <no associated attribute callbacks>

    context: dt 0xfffffb6093ccead20 Acx01000!AfxDriver (size is 0x20 bytes)
        EvtCleanupCallback ffffff8038a455780
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_*
*,Acx::AfxDriver>::EvtObjectContextCleanupThunk
        EvtDestroyCallback ffffff8038a455740
    Acx01000!Acx::WdfCpp::ObjectContext<WDFDRIVER_*
*,Acx::AfxDriver>::EvtObjectContextDestroyThunk

Child WDFHANDLES of 0x000049f6bfe9d5d8:
    !wdfhandle 0x000049f6bfe9d5d8  dt FxDriver 0xfffffb60940162a20 Context
fffffb60940162bc0, Context fffffb6093ccead20 Cleanup ffffff8038a455780 Destroy
ffffff8038a455740
        !wdfdevice 0x000049f6bffba488  dt FxDevice 0xfffffb60940045b70
    Context fffffb60940045e60 Cleanup ffffff8038a3e3c90, Context fffffb60933f030f0
    Cleanup ffffff8038a451910 Destroy ffffff8038a451740, Context fffffb60940a26b90,
    Context fffffb609409f2e00
        WDF INTERNAL      dt FxDefaultIrpHandler 0xfffffb6093fa54130
        WDF INTERNAL      dt FxPkgGeneral 0xfffffb609401f5610
        WDF INTERNAL      dt FxWmiIrpHandler 0xfffffb609401f6510
        WDF INTERNAL      dt FxPkgIo 0xfffffb6093f6d2400
            !wdfqueue 0x000049f6c00114b8  dt FxIoQueue
0xfffffb6093ffeeb40
        WDF INTERNAL      dt FxPkgFdo 0xfffffb6093f2ae020
            !wdfhandle 0x000049f6bfe099f8  dt FxCmResList 0xfffffb609401f6600
            !wdfhandle 0x000049f6bfe0a358  dt FxCmResList 0xfffffb609401f5ca0
            !wdfchildlist 0x000049f6c09adb98  dt FxChildList
0xfffffb6093f652460
            !wdfiotarget 0x000049f6c092d1d8  dt FxIoTarget
```

```

0xfffffb6093f6d2e20
    !wdfqueue 0x000049f6c04904c8  dt FxIoQueue 0xfffffb6093fb6fb30
Context fffffb6093fa8dc0 Cleanup fffff8038a45a350 Destroy fffff8038a45a310
        !wdfhandle 0x000049f6c09e7ed8  dt FxWorkItem 0xfffffb6093f618120
Context fffffb6093f618220
    WDF INTERNAL  dt FxWmiProvider 0xfffffb609403d9b50
        WDF INTERNAL  dt FxWmiInstanceExternal 0xfffffb60940a11320
        WDF INTERNAL  dt FxWmiProvider 0xfffffb609403d9300
        WDF INTERNAL  dt FxWmiInstanceExternal 0xfffffb60940a11720
    !wdfdevice 0x000049f6bef1a848  dt FxDevice 0xfffffb609410e57b0
Context fffffb609410e5aa0 Cleanup fffff8038a3e6400, Context fffffb60933f090d0
Cleanup fffff8038a451910 Destroy fffff8038a451740
    WDF INTERNAL  dt FxDefaultIrpHandler 0xfffffb60940c16450
    WDF INTERNAL  dt FxPkgGeneral 0xfffffb60940bc9a10
    WDF INTERNAL  dt FxWmiIrpHandler 0xfffffb60940bc9380
    WDF INTERNAL  dt FxPkgIo 0xfffffb609400ab5c0
        !wdfqueue 0x000049f6beef588  dt FxIoQueue
0xfffffb60941102a70
    WDF INTERNAL  dt FxPkgPdo 0xfffffb609410e6020
    !wdfhandle 0x000049f6bf436e58  dt FxCmResList 0xfffffb60940bc91a0
    !wdfhandle 0x000049f6bf436d68  dt FxCmResList 0xfffffb60940bc9290
    !wdfqueue 0x000049f6bef185c8  dt FxIoQueue 0xfffffb609410e7a30
Context fffffb60940e72ed0 Cleanup fffff8038a45a350 Destroy fffff8038a45a310
        !wdfhandle 0x000049f6bf436b88 [ACXCIRCUIT]  dt FxUserObject
0xfffffb60940bc9470 Context fffffb60940bc9520, Context
...

```

WDF log commands

[!wdflogdump](#) can be useful for troubleshooting an ACX driver by displaying WDF log information.

Display the log for ACX using the -d driver option.

```

dbgcmd

0: kd> !wdflogdump acx01000 -d
Log dump command
=====
!wdflogdump Acx01000 -a 0xFFFF820527861000 AFX_Client1      4096
!wdflogdump Acx01000 -a 0xFFFF8205215C0000 AFX_Log          4096
Trace searchpath is:

Trace format prefix is: %7!u!: %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\Dbg\sym\Acx01000.pdb\B13D39B43205B60C07935803D7CB96981\Acx010
00.pdb
--- start of log ---
AFX_Client1 1: Acx::AfxCircuit::Register - INFO:ACXCIRCUIT 00007DFAD9CCCCB8
Registered
AFX_Client1 2: Acx::AfxCircuit::PowerUpNotification - INFO:ACXCIRCUIT

```

```
00007DFAD9CCCCB8, EvtAcxCircuitPowerUp callback, STATUS_SUCCESS
AFX_Client1 5: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 7: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 8: Acx::AfxMute::EvtMuteEventEnableCallback - INFO:ACXMUTE
00007DFADF997A18, enabled ACXEVENT 00007DFADF996F98
AFX_Client1 10: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 12: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 13: Acx::AfxPin::EvtJackEventEnableCallback - INFO:ACXPIN
00007DFAD4697238, enabled ACXEVENT 00007DFADF9973F8
AFX_Client1 15: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
AFX_Client1 17: Acx::AfxPin::GetModesCount - WARN:ACXPIN 00007DFAD4697238,
failed to get default format for processing mode 9e90ea20-b493-4fd1-a1a8-
7e1361a956cf, 0xc0000225(STATUS_NOT_FOUND)
```

Use [!wdflogdump](#) to display the framework log for a specific ACX driver using the **-f** option.

```
dbgcmd

0: kd> !wdflogdump AcxHdAudio -f
Trace searchpath is:

Trace format prefix is: %7!u!: %4!s! %!FUNC! -
Trying to extract TMF information from -
C:\ProgramData\Txt\sym\Wdf01000.pdb\CBDEA3A4F64C17C1752E652A91DD14761\Wdf010
00.pdb
Gather log: Please wait, this may take a moment (reading 4024 bytes).
% read so far ... 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
There are 65 log entries
--- start of log ---
16131: 09/03/2023-23:43:07.3233594 imp_WdfRegistryOpenKey - new WDFKEY
object open failed, 0xc0000034(STATUS_OBJECT_NAME_NOT_FOUND)
16132: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleDecrementIo from FxIdleBusy
16133: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleStartTimer from FxIdleDecrementIo
16134: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleTimerRunning from FxIdleStartTimer
16135: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
```

```
state FxIdleCancelTimer from FxIdleTimerRunning
16136: 09/03/2023-23:43:07.3233594 FxPowerIdleMachine::ProcessEventLocked -
WDFDEVICE 0x00007DFADB0A5358 !devobj 0xFFFF820521608AF0 entering power idle
state FxIdleCheckIoCount from FxIdleCancelTimer
16137:
...
...
```

See also

For more information, see [Kernel Streaming Debugging](#). For a walkthrough of debugging with a WDM audio driver, see [Debug Drivers - Step by Step Lab \(Sysvad Kernel Mode\)](#). For more information about ACX, see [ACX audio class extensions overview](#).

Feedback

Was this page helpful?



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

SCSI Miniport Extensions (Scsikd.dll and Minipkd.dll)

Article • 10/25/2023

Extension commands that are useful for debugging SCSI miniport drivers can be found in Scsikd.dll and Minipkd.dll.

You can use the Scsikd.dll extension commands with any version of Windows. However, you can only use the Minipkd.dll extension commands with Windows XP and later versions of Windows. Commands in Minipkd.dll are only applicable to SCSIport-based miniports.

For more information, see [SCSI Miniport Debugging](#).

!scsikd.help

Article • 04/03/2024

The **!scsikd.help** extension displays help text for Scsikd.dll extension commands.

```
dbgcmd
!scsikd.help
```

DLL

Scsikd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!scsikd.classext

Article • 10/25/2023

The **!scsikd.classext** extension displays the specified class Plug and Play (PnP) device.

dbgcmd

```
!scsikd.classext [Device [Level]]
```

Parameters

Device

Specifies the device object or device extension of a class PnP device. If *Device* is omitted, a list of all class PnP extensions is displayed.

Level

Specifies the amount of detail to display. This parameter can take 0, 1, or 2 as values, with 2 giving the most detail. The default is 0.

DLL

Windows 2000	Scsikd.dll
Windows XP and later	Scsikd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

Here is an example of the **!scsikd.classext** display:

dbgcmd

```
0: kd> !scsikd.classext
' !scsikd.classext 8633e3f0 ' ( ) "IBM" "/ "DDYS-T09170M
" / "S93E" / " XBY45906"
' !scsikd.classext 86347b48 ' (paging device) "IBM" "/ "DDYS-T09170M
" / "S80D" / " VDA60491"
' !scsikd.classext 86347360 ' ( ) "UNISYS" "/
```

```
"003451ST34573WC " / "5786" / "HN0220750000181300L6"  
' !scsikd.classext 861d1898 ' ( ) "" / "MATSHITA CD-ROM CR-  
177" / "7T03" / "
```

```
usage: !classext <class fdo> <level [0-2]>
```

!scsikd.scsiext

Article • 04/03/2024

The **!scsikd.scsiext** extension displays detailed information about the specified SCSI port extension.

```
dbgcmd
```

```
!scsikd.scsiext Device
```

Parameters

Device

Specifies the device object or device extension of a SCSI port extension.

DLL

Scsikd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

Here is an example of the **!scsikd.scsiext** display, where the SCSI port extension has been specified by a functional device object (FDO); this can be obtained from the **DO** field or **DevExt** field in the [!minipkd.adapters](#) display:

```
dbgcmd
```

```
kd> !scsikd.scsiext 816f9a40
Scsiport functional device extension at address 816f9af8
Common Extension:
    Initialized
    DO 0x816f9a40  LowerObject 0x816e8030  SRB Flags 00000000
    Current Power (D0,S0)  Desired Power D-1 Idle 00000000
    Current PnP state 0x0    Previous state 0xff
    DispatchTable f9aee200  UsePathCounts (P0, H0, C0)
Adapter Extension:
    Port 2      IsPnp VirtualSlot HasInterrupt
```

```

LowerPdo 0x816e8030 HwDevExt 0x8170a004 Active Requests 0xffffffff
MaxBus 0x01 MaxTarget 0x10 MaxLun 0x08
Port Flags (0x00001000): PD_DISCONNECT_RUNNING
NonCacheExt 0x81702000 IoBase 0x00002000 Int 0x1a
RealBus# 0x0 RealSlot# 0x2
Timeout 0xffffffff DpcFlags 0x00000000 Sequence 0x00000003
Srb Ext Header 0x817061a0 No. Requests 0x00000012
QueueTag BitMap 0x00000000 Hint 0x00000000
MaxQueueTag 0xfe (@0x816f9c58)
LuExt Size 0x00000038 SrbExt Size 0x00000188
SG List Size - Small 17 Large 0
Emergency - SrbData 0x816f9830 Blocked List @0x816f9e94
CommonBuff - Size: 0x00006000 PA: 0x0000000001702000 VA: 0x81702000
Ke Objects - Int1: 0x8175ba50 Int2: 0x00000000 Dma: 0x816f9340
Lookaside - SrbData @ 0x816f9e40 SgList @0x00000000 Remove:
@0x00000000
Resources - Raw: 0x817ba190 Translated: 0x81709678
Port Config 8177fde8
DeviceMap Handles: Port 0000009c Busses e12d7b38
Interrupt Data @0x816f9ce4:
Flags (0x00000000):
Ready LUN 0x00000000 Wmi Events 0x00000000
Completed Request List (@0x816f9ce8): 0 entries
LUN 816ea0e8 @ ( 0, 1, 0) c ev pnp(00/ff) pow(0 ,0) DevObj 816ea030

```

Here is an example of the **!scsikd.scsiext** display, where the SCSI port extension has been specified by a physical device object (PDO); this can be obtained from the **DevObj** field or **LUN** field in the **!minipkd.adapters** display:

```

dbgcmd

kd> !scsikd.scsiext 816ea030
Scsiport physical device extension at address 816ea0e8
Common Extension:
    Initialized
    DO 0x816ea030 LowerObject 0x816f9a40 SRB Flags 00000000
    Current Power (D0,S0) Desired Power D-1 Idle 0x8176c780
    Current PnP state 0x0 Previous state 0xff
    DispatchTable f9aeee180 UsePathCounts (P0, H0, C0)
Logical Unit Extension:
    Address (2, 0, 1, 0) Claimed Enumerated Visible
    LuFlags (0x00000000):
        Retry 0x00 Key 0x00000000
        Lock 0x00000000 Pause 0x00000000 CurrentLock: 0x00000000
        HwLuExt 0x8177ce10 Adapter 0x816f9af8 Timeout 0xffffffff
        NextLun 0x00000000 ReadyLun 0x00000000
        Pending 0x00000000 Busy 0x00000000 Untagged 0x00000000
        Q Depth 000 (1628450047) InquiryData 0x816ea206
        DeviceMap Keys: Target 0x0000d0 Lun 00000000
        Bypass SRB_DATA blocks 4 @ 816ea270 List 816ea810
        RS Irp 8177dd80 Srb @ 816eaa0c MDL @ 816eaa4c
        Request List @0x816ea1f0 is empty

```

Feedback

Was this page helpful?

 Yes

 No

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

!scsikd.srbdata

Article • 04/03/2024

The **!scsikd.srbdata** extension displays the specified SRB_DATA tracking block.

```
dbgcmd
!scsikd.srbdata Address
```

Parameters

Address

Specifies the address of an SRB_DATA tracking block.

DLL

Scsikd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!minipkd.help

Article • 04/03/2024

The **!minipkd.help** extension displays help text for the Minipkd.dll extension commands.

```
dbgcmd
```

```
!minipkd.help
```

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

If an error message similar to the following appears, it indicates that the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols.

```
dbgcmd
```

```
minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

Use the [.sympath \(Set Symbol Path\)](#) command to display the current path and change the path. Use the [.reload \(Reload Module\)](#) command to reload symbols from the current path.

Feedback

Was this page helpful?

 Yes

 No

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

!minipkd.adapter

Article • 04/03/2024

The **!minipkd.adapter** extension displays information about the specified adapter.



Parameters

Address

Specifies the address of an adapter.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

The address of an adapter can be found in the **DevExt** field of the [!minipkd.adapters](#) display.

Feedback

Was this page helpful?

Yes

No

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

!minipkd.adapters

Article • 04/03/2024

The **!minipkd.adapters** extension displays all of the adapters that work with the SCSI Port driver that have been identified in the system, and the individual devices associated with each adapter.

dbgcmd

!minipkd.adapters

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

The display includes the driver name, the device object address, and the device extension address for each adapter. The display for each adapter also includes a list of each device on the adapter. The display for each device includes the device extension address, the SCSI address, the device object address, and some flags for the device. Information about the Plug and Play state and the power state is also included.

The flags in the display are explained in the following table:

[+] Expand table

Flag	Meaning
c	Claimed. Indicates that the device has a driver on it.
m	Missing. Indicates that the device was present on the bus in a prior scan but was not present during the latest scan.

Flag	Meaning
e	Enumerated. Indicates that the device has been reported to the Plug and Play manager.
v	Visible. Indicates that the device has been enumerated by the system. This flag is more significant when it is not present for a device.
p	Paging. Indicates that the device is in the paging path.
d	Dump. Indicates that the device is in the crash dump path and will be used for a crash dump.
h	Hibernate. Indicates that the device is hibernating.

Here is an example of the **!minipkd.adapters** display:

```
dbgcmd

0: kd> !minipkd.adapters
Adapter \Driver\lp6nds35      DO 86334a70          DevExt 86334b28
Adapter \Driver\adpu160m      DO 8633da70          DevExt 8633db28
LUN 862e60f8 @ (0,0,0) c ev    pnp(00/ff) pow(0,0) DevObj 862e6040
LUN 863530f8 @ (0,1,0) c ev p d pnp(00/ff) pow(0,0) DevObj 86353040
LUN 862e50f8 @ (0,2,0) c ev    pnp(00/ff) pow(0,0) DevObj 862e5040
LUN 863520f8 @ (0,6,0)   ev    pnp(00/ff) pow(0,0) DevObj 86352040
Adapter \Driver\adpu160m      DO 86376040          DevExt 863760f8
```

An error message similar to the following indicates that either the symbol path is incorrect and does not point to the correct version of the Scsiport.sys symbols, or that Windows has not identified any adapters that work with the SCSI Port driver.

```
dbgcmd

minipkd error (0) <path> ... \minipkd\minipkd.c @ line 435
```

If the **!minipkd.help** extension command returns help information successfully, the SCSI Port symbols are correct.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!minipkd.exports

Article • 04/03/2024

The **!minipkd.exports** extension displays the addresses of the miniport exports for the specified adapter.

```
dbgcmd
```

```
!minipkd.exports Adapter
```

Parameters

Adapter

Specifies the address of an adapter.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!minipkd.lun

Article • 04/03/2024

The **!minipkd.lun** extension displays detailed information about the specified Logical Unit Extension (LUN).

dbgcmd

```
!minipkd.lun LUN  
!minipkd.lun Device
```

Parameters

LUN

Specifies the address of the LUN.

Device

Specifies the physical device object (PDO) for the LUN.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

A LUN is typically referred to as a *device*. Thus, this extension displays information about a device on an adapter.

The LUN can be specified either by its address (which can be found in the **LUN** field of the [!minipkd.adapters](#) display), or by its physical device object (which can be found in the **DevObj** field of the [!minipkd.adapters](#) display).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

!minipkd.portconfig

Article • 04/03/2024

The **!minipkd.portconfig** extension displays information about the specified PORT_CONFIGURATION_INFORMATION data structure.

dbgcmd

```
!minipkd.portconfig PortConfig
```

Parameters

PortConfig

Specifies the address of a PORT_CONFIGURATION_INFORMATION data structure.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

The *PortConfig* address can be found in the **Port Config Info** field of the [!minipkd.adapter](#) display.

Feedback

Was this page helpful?

 Yes

 No

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

!minipkd.req

Article • 04/03/2024

The **!minipkd.req** extension displays information about all of the currently active requests on the specified adapter or device.

dbgcmd

```
!minipkd.req Adapter  
!minipkd.req Device
```

Parameters

Adapter

Specifies the address of the adapter.

Device

Specifies the physical device object (PDO) for the Logical Unit Extension (LUN) device.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

The PDO for a LUN can be found in the **DevObj** field of the [!minipkd.adapters](#) display.

Feedback

Was this page helpful?

 Yes

 No

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

!minipkd.srb

Article • 04/03/2024

The **!minipkd.srb** extension displays the specified SCSI request block (SRB) data structure.

dbgcmd

```
!minipkd.srb SRB
```

Parameters

SRB

Specifies the address of an SRB.

DLL

Minipkd.dll

Additional Information

For more information, see [SCSI Miniport Debugging](#).

Remarks

The addresses of all currently active requests can be found in the *SRB* fields of the output from the **!minipkd.req** command.

This extension displays the status of the SRB, the driver it is addressed to, the SCSI that issued the SRB and its address, and a hexadecimal flag value. If 0x10000 is set in the flag value, this request is currently in the miniport.

Feedback

Was this page helpful?

 Yes

 No

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

Windows Driver Framework Extensions (Wdfkd.dll)

Article • 10/25/2023

Extension commands that are useful for debugging drivers built with the Kernel-Mode Driver Framework (KMDF) or version 2 of the User-Mode Driver Framework (UMDF 2) are implemented in Wdfkd.dll.

These extensions can be used on Microsoft Windows XP and later operating systems. Some extensions have additional restrictions; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

For more information about how to use these extensions, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.help

Article • 10/25/2023

The **!wdfkd.help** extension displays help information about all Wdfkd.dll extension commands.

```
dbgcmd
```

```
!wdfkd.help
```

DLL

Wdfkd.dll

Additional Information

The **!wdfkd.help** extension is equivalent to the [!wdfkd.wdfhelp](#) extension.

For more information about debugging framework-based drivers, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfchildlist

Article • 10/25/2023

The **!wdfkd.wdfchildlist** extension displays a child list's state and information about all of the device identification descriptions that are in the child list.

```
dbgcmd
```

```
!wdfkd.wdfchildlist Handle
```

Parameters

Handle

A WDFCHILDLIST-typed handle to the child list.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example shows a **!wdfkd.wdfchildlist** display.

```
dbgcmd
```

```
kd> !wdfchildlist 0x7cc090c8
## Dumping WDFCHILDLIST 0x7cc090c8
-----
owning !WDFDEVICE 0x7ca7b1c0
ID description size 0x8

State:
```

```
-----  
List is unlocked, changes will be applied immediately  
No scans or enumerations are active on the list
```

Descriptions:

```
-----  
PDO !WDFDEVICE 0x7cad31c8, ID description 0x83ac4ff4  
+Device WDM !devobj 0x81fb00e8, WDF pnp state WdfDevStatePnpStarted (0x119)  
+Device found in last scan
```

No pending insertions are in the list.

Callbacks:

```
-----  
EvtChildListCreateDevice: wdfrawbusenumtest!RawBus_RawPdo_Create  
(f22263b0)
```

!wdfkd.wdfcollection

Article • 10/25/2023

The **!wdfkd.wdfcollection** extension displays all of the objects that are stored in a WDFCOLLECTION structure.

dbgcmd

```
!wdfkd.wdfcollection Handle
```

Parameters

Handle

A WDFCOLLECTION-typed handle to the structure.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfcommonbuffer

Article • 10/25/2023

The **!wdfkd.wdfcommonbuffer** extension displays information about a WDF common buffer object.

dbgcmd

!wdfkd.wdfcommonbuffer Handle

Parameters

Handle

A handle to a framework common buffer object (WDFCOMMONBUFFER).

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfcrashdump

Article • 10/25/2023

The `!wdfkd.wdfcrashdump` extension displays error log information and other crash dump information from a minidump file, if the data is present.

KMDF

```
dbgcmd
!wdfkd.wdfcrashdump [InfoType]
```

UMDF

```
dbgcmd
!wdfkd.wdfcrashdump [DriverName.dll][-d | -f | -m]
```

Parameters

InfoType

Specifies the kind of information to display. *InfoType* is optional and can be any one of the following values:

log

Displays error log information, if available in the crash dump file. This is the default value.

loader

Displays the minidump's dynamic-bound drivers.

DriverName.dll

Specifies the name of a UMDF driver. You must include the .dll file suffix. If this optional parameter is omitted, output includes metadata, the loaded module list, and available logs.

-d

Displays only the driver logs.

-f

Displays only the framework logs.

-m

Merges framework and driver logs in their recorded order.

DLL

Wdfkd.dll

Frameworks

KMDF

UMDF 2.15

Remarks

This example shows how to use `!wdfkd.wdfcrashdump` to view information about KMDF drivers. If you specify `loader` for `InfoType`, the output includes dynamic-bound drivers in the minidump file.

```
dbgcmd

0: kd> !wdfcrashdump loader
Retrieving crashdump loader information...
## Local buffer 0x002B4D00, bufferSize 720
-----
  ImageName      Version      FxGlobals
  Wdf01000        v1.9(6902)
  msisadrv       v1.9(6913) 0x84deb260
  vdrvroot       v1.9(6913) 0x860e8260
  storflt        v1.5(6000) 0x861dfe90
  cdrom          v1.9(6913) 0x84dca008
  intelppm       v1.9(6913) 0x864704a8
  HDAudBus       v1.7(6001) 0x86101c98
  1394ohci       v1.7(6001) 0x8610d2e8
  CompositeBus   v1.9(6913) 0x86505b98
  ObjTestClassExt v1.9(6902) 0x865b7f00
  mqfilter        v1.9(6902) 0x865b8008
  mqueue          v1.9(6902) 0x865b6910
  umbus           v1.9(6913) 0x8618aea0
  monitor         v1.9(6913) 0x86aac1d8
  PEAUTH          v1.5(6000) 0x854e5350
-----
```

This example shows how to use `!wdfkd.wdfcrashdump` to view information about UMDF drivers. If you issue `!wdfkd.wdfcrashdump` with no parameters, the output

includes the driver that caused the crash and a list of all loaded drivers in the host process that failed. You can click on drivers in this list that have associated logs.

```
dbgcmd

0:001> !wdfkd.wdfcrashdump
Opening minidump at location C:\temp\WudfHost_ext_1312.dmp

Faulting driver: wpptest.dll
Failure type: Unhandled Exception (WUDFUnhandledException)
Faulting thread ID: 2840

Listing all drivers loaded in this host process at the time of the failure:

ServiceName
wpptest
CoverageCx0102
coverage
WUDFVhidmini
ToastMon
WUDFOsrUsbFilter
```

In the example above, output includes failure type, which is the event type in the WER report. Here, it can be **WUDFVerifierFailure** or **WUDFUnhandledException**. For more information, see [Accessing UMDF Metadata in WER Reports](#). The output for UMDF includes an error code, if event type is **WUDFVerifierFailure**.

To display the framework's error log records from a [complete memory dump](#), a [kernel memory dump](#), or a [live kernel-mode target](#), you can also try the [**!wdfkd.wdflogdump**](#) extension.

Additional Information

For information about enabling the inflight trace recorder for your driver, see [Using Inflight Trace Recorder \(IFR\) in KMDF and UMDF 2 Drivers](#). For more information about debugging WDF drivers, see [Debugging WDF Drivers](#). For information about KMDF debugging, see [Kernel-Mode Driver Framework Debugging](#).

See also

[**!wdfkd.wdflogdump**](#)

[**!wdfkd.wdfsettraceprefix**](#)

!wdfkd.wdfdevext

Article • 10/25/2023

The **!wdfkd.wdfdevext** extension displays information that is associated with the **DeviceExtension** member of a Microsoft Windows Driver Model (WDM) DEVICE_OBJECT structure.

```
dbgcmd
```

```
!wdfkd.wdfdevext DeviceExtension
```

Parameters

DeviceExtension

A pointer to a device extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

Here is an example for HdAudBus.sys, which is a KMDF driver. Use **!devnode** to find a device node that has HdAudBus as its function driver. Take the physical device object (PDO) from the output and pass it to **!devstack**. Take the device extension address from the output of **!devstack** and pass it to **!wdfdevext**.

```
dbgcmd
```

```
0: kd> !devnode 0 1 hdaudbus
Dumping IopRootDeviceNode (= 0xfffffe000002cf30)
DevNode 0xfffffe000009b7a50 for PDO 0xfffffe00000226880
```

```

InstancePath is
"PCI\VEN_8086&DEV_293E&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D8"
  ServiceName is "HDAudBus"
  ...
0: kd> !devstack 0xfffffe00000226880
  !DevObj          !DrvObj          !DevExt          ObjectName
  fffffe00001351e20  \Driver\HDAudBus  fffffe000009a3c00
> fffffe00000226880  \Driver\pci      fffffe000002269d0  NTPNP_PCI0009
!DevNode fffffe000009b7a50 :
  DeviceInst is
"PCI\VEN_8086&DEV_293E&SUBSYS_2819103C&REV_02\3&33fd14ca&0&D8"
  ServiceName is "HDAudBus"
0: kd> *
0: kd> !wdfdevext fffffe000009a3c00
Device context is 0xfffffe000009a3c00
  context: dt 0xfffffe000009a3c00 HDAudBus!HDAudioDeviceExtension (size is
0xa8 bytes)
  EvtCleanupCallback fffff80001f35950
HDAudBus!HdAudBusEvtDeviceCleanupCallback

!wdfdevice 0x00001fffff65c6e8
!wdfobject 0xfffffe000009a3910

```

Here is an example for Wudfrd.sys, which is the function driver for the kernel-mode portion of a UMDF 2 driver stack. Use **!devnode** to find a device node that has Wudfrd as its function driver. Take the physical device object (PDO) from the output and pass it to **!devstack**. Take the device extension address from the output of **!devstack** and pass it to **!wdfdevext**.

```

dbgcmd

0: kd> !devnode 0 1 wudfrd
Dumping IopRootDeviceNode (= 0xfffffe000002cf30)
DevNode 0xfffffe00000a1e530 for PDO 0xfffffe00000b15b00
  InstancePath is "ROOT\SAMPLE\0001"
  ServiceName is "WUDFRd"
  ...
0: kd> !devstack 0xfffffe00000b15b00
  !DevObj          !DrvObj          !DevExt          ObjectName
  fffffe00000c11040  \Driver\WUDFRd  fffffe00000c11190
> fffffe00000b15b00  \Driver\PnpManager 00000000  00000052
!DevNode fffffe00000a1e530 :
  DeviceInst is "ROOT\SAMPLE\0001"
  ServiceName is "WUDFRd"
0: kd> *
0: kd> !wdfdevext fffffe00000c11190
## Device context is 0xfffffe00000c11190

##  UMDF Device Instances for this Redirector extension

DriverManagerProcess: 0xfffffe00003470500

```

ImageName	Ver	DevStack	HostProcess
DeviceID			
MyUmdf2Driver.dll	v2.0	0x000000a5a3ab5f70	0xfffffe00000c32900
\Device\00000052			

!wdfkd.wdfdevice

Article • 10/25/2023

The **!wdfkd.wdfdevice** extension displays information that is associated with a WDFDEVICE-typed object handle.

dbgcmd

```
!wdfkd.wdfdevice Handle [Flags]
```

Parameters

Handle

A handle to a WDFDEVICE-typed object.

Flags

Optional. The kind of information to display. *Flags* can be any combination of the following bits:

Bit 0 (0x1)

The display will include verbose information about the device, such as the associated WDFCHILDLIST-typed handles, synchronization scope, and execution level.

Bit 1 (0x2)

The display will include detailed power state information.

Bit 2 (0x4)

The display will include detailed power policy state information.

Bit 3 (0x8)

The display will include detailed Plug and Play (PnP) state information.

Bit 4 (0x10)

The display will include the device object's callback functions.

DLL

Wdfkd.dll

Frameworks

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example uses the `!wdfkd.wdfdevice` extension on a WDFDEVICE handle that represents a physical device object (PDO), without specifying any flags.

```
dbgcmd

kd> !wdfdevice 0x7cad31c8

# Dumping WDFDEVICE 0x7cad31c8
=====

WDM PDEVICE_OBJECTs: self 81fb00e8

Pnp state: 119 ( WdfDevStatePnpStarted )
Power state: 31f ( WdfDevStatePowerDx )
Power Pol state: 508 ( WdfDevStatePwrPolWaitingUnarmed )

Parent WDFDEVICE 7ca7b1c0
Parent states:
    Pnp state: 119 ( WdfDevStatePnpStarted )
    Power state: 307 ( WdfDevStatePowerD0 )
    Power Pol state: 565 ( WdfDevStatePwrPolStarted )

No pended pnp or power irps
Device is the power policy owner for the stack
```

The following example displays the same device object as the preceding example, but this time with a flag value of 0xF. This flag value, a combination of the bits 0x1, 0x2, 0x4, and 0x8, causes the display to include verbose device information, power state information, power policy state information, and PnP state information.

```
dbgcmd

kd> !wdfdevice 0x7cad31c8 f

# Dumping WDFDEVICE 0x7cad31c8
=====

WDM PDEVICE_OBJECTs: self 81fb00e8

Pnp state: 119 ( WdfDevStatePnpStarted )
```

```
Power state: 31f ( WdfDevStatePowerDx )
Power Pol state: 508 ( WdfDevStatePwrPolWaitingUnarmed )
```

```
Parent WDFDEVICE 7ca7b1c0
```

```
Parent states:
```

```
Pnp state: 119 ( WdfDevStatePnpStarted )
Power state: 307 ( WdfDevStatePowerD0 )
Power Pol state: 565 ( WdfDevStatePwrPolStarted )
```

```
No pended pnp or power irps
```

```
Device is the power policy owner for the stack
```

```
Pnp state history:
```

```
[0] WdfDevStatePnpObjectCreated (0x100)
[1] WdfDevStatePnpInit (0x105)
[2] WdfDevStatePnpInitStarting (0x106)
[3] WdfDevStatePnpHardwareAvailable (0x108)
[4] WdfDevStatePnpEnableInterfaces (0x109)
[5] WdfDevStatePnpStarted (0x119)
```

```
Power state history:
```

```
[0] WdfDevStatePowerD0StartingConnectInterrupt (0x310)
[1] WdfDevStatePowerD0StartingDmaEnable (0x311)
[2] WdfDevStatePowerD0StartingStartSelfManagedIo (0x312)
[3] WdfDevStatePowerDecideD0State (0x313)
[4] WdfDevStatePowerD0BusWakeOwner (0x309)
[5] WdfDevStatePowerGotoDx (0x31a)
[6] WdfDevStatePowerGotoDxIoStopped (0x31c)
[7] WdfDevStatePowerDx (0x31f)
```

```
Power policy state history:
```

```
[0] WdfDevStatePwrPolStarting (0x501)
[1] WdfDevStatePwrPolStartingSucceeded (0x502)
[2] WdfDevStatePwrPolStartingDecideS0Wake (0x504)
[3] WdfDevStatePwrPolStartedIdleCapable (0x505)
[4] WdfDevStatePwrPolTimerExpiredNoWake (0x506)
[5] WdfDevStatePwrPolTimerExpiredNoWakeCompletePowerDown (0x507)
[6] WdfDevStatePwrPolWaitingUnarmedQueryIdle (0x509)
[7] WdfDevStatePwrPolWaitingUnarmed (0x508)
```

```
WDFCHILDLIST Handles:
```

```
!WDFCHILDLIST 0x7ce710c8
```

```
SynchronizationScope is WdfSynchronizationScopeNone
```

```
ExecutionLevel is WdfExecutionLevelDispatch
```

!wdfkd.wdfdeviceinterrupts

Article • 10/25/2023

The **!wdfkd.wdfdeviceinterrupts** extension displays all the interrupt objects for a specified device handle.

dbgcmd

```
!wdfkd.wdfdeviceinterrupts Handle
```

Parameters

Handle

A handle to a WDFDEVICE-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfdevicequeues

Article • 10/25/2023

The **!wdfkd.wdfdevicequeues** extension displays information about all of the framework queue objects that belong to a specified device.

dbgcmd

```
!wdfkd.wdfdevicequeues Handle
```

Parameters

Handle

A handle to a WDFDEVICE-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#) and [!wdfkd.wdfqueue](#).

Remarks

The following example shows the display from the **!wdfkd.wdfdevicequeues** extension.

dbgcmd

```
kd> !wdfdevicequeues 0x7cad31c8
# Dumping queues of WDFDEVICE 0x7cad31c8
=====
## Number of queues: 3
-----
Queue: 1 (!wdfqueue 0x7d67d1e8)
```

```
    Manual, Not power-managed, PowerOn, Can accept, Can dispatch,  
ExecutionLevelDispatch, SynchronizationScopeNone
```

```
    Number of driver owned requests: 0
```

```
    Number of waiting requests: 0
```

```
## This is WDF internal queue for create requests.
```

```
-----  
Queue: 2 (!wdfqueue 0x7ce7d1e8)
```

```
    Parallel, Power-managed, PowerOff, Can accept, Can dispatch,  
ExecutionLevelDispatch, SynchronizationScopeNone
```

```
    Number of driver owned requests: 0
```

```
    Number of waiting requests: 0
```

```
##      EvtIoDefault: (0xf221fad0) wdfrawbusenumtest!EvtIoQueueDefault
```

```
-----  
Queue: 3 (!wdfqueue 0x7cd671e8)
```

```
    Parallel, Power-managed, PowerOff, Can accept, Can dispatch,  
ExecutionLevelDispatch, SynchronizationScopeNone
```

```
    Number of driver owned requests: 0
```

```
    Number of waiting requests: 0
```

```
    EvtIoDeviceControl: (0xf2226ac0)
```

```
    wdfrawbusenumtest!RawBus_RawPdo_EvtDeviceControl
```

!wdfkd.wdfdmaenabler

Article • 10/25/2023

The **!wdfkd.wdfdmaenabler** extension displays information about a WDF direct memory access (DMA) object, and its transaction and common buffer objects.

dbgcmd

!wdfkd.wdfdmaenabler Handle

Parameters

Handle

A handle to a framework DMA enabler object (WDFDMAENABLER).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfdmaenablers

Article • 10/25/2023

The `!wdfkd.wdfdmaenablers` extension displays all WDF direct memory access (DMA) objects associated with a specified device object. It also displays their associated transaction and common buffer objects.

dbgcmd

```
!wdfkd.wdfdmaenablers Handle
```

Parameters

Handle

A handle to a framework device object (WDFDEVICE).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfmatransaction

Article • 10/25/2023

The **!wdfkd.wdfmatransaction** extension displays information about a WDF direct memory access (DMA) transaction object.

dbgcmd

!wdfkd.wdfmatransaction Handle

Parameters

Handle

A handle to a framework DMA transaction object (WDFDMA TRANSACTION).

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfdriverinfo

Article • 10/25/2023

The **!wdfkd.wdfdriverinfo** extension displays information about the specified driver, including its device tree, the version of the Kernel-Mode Driver Framework (KMDF) library that the driver was compiled with, and a list of the framework device objects that the driver created.

dbgcmd

```
!wdfkd.wdfdriverinfo [DriverName [Flags]]
```

Parameters

DriverName

Optional. The name of the driver. *DriverName* must not include the .sys file name extension.

Flags

Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits:

Bit 0 (0x1)

The display will include verifier settings for the driver, and will also include a count of WDF objects. This flag can be combined with bit 6 (0x40) to display internal objects.

Bit 4 (0x10)

The display will include the KMDF handle hierarchy for the driver.

Bit 5 (0x20)

The display will include context and callback function information for each handle. This flag is valid only when bit 4 (0x10) is set.

Bit 6 (0x40)

The display will include additional information for each handle. This flag is valid only when bit 4 (0x10) is set. This flag can be combined with bit 0 (0x1) to display internal objects.

Bit 7 (0x80)

The handle information will be displayed in a more compact format.

Bit 8 (0x100)

The display will left align internal type information. This flag is valid only when bit 4 (0x10) is set.

Bit 9 (0x200)

The display will include handles that the driver potentially leaked. KMDF version 1.1 and later support this flag. This flag is valid only when bit 4 (0x10) is set.

Bit 10 (0x400)

The display will include the device tree in verbose form.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

If you omit the *DriverName* parameter, the default driver is used. You can display the default driver by using the [!wdfkd.wdfgetdriver](#) extension; you can set the default driver by using the [!wdfkd.wdfsetdriver](#) extension.

The following example shows the display from the [!wdfkd.wdfdriverinfo](#) extension.

```
dbgcmd

## kd> !wdfdriverinfo wdfrawbusenumtest
-----
Default driver image name:    wdfrawbusenumtest
WDF library image name:      Wdf01000
FxDriverGlobals  0x83b7af18
WdfBindInfo      0xf22250ec
##    Version        v1.5 build(1234)
-----
WDFDRIVER: 0x7cbc90d0

!WDFDEVICE 0x7ca7b1c0
    context: dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
```

```
<no associated attribute callbacks>

!WDFDEVICE 0x7cad31c8
    context: dt 0x8352cff0 RAW_PDO_CONTEXT (size is 0xc bytes)
    <no associated attribute callbacks>
```

!wdfkd.wdfextendwatchdog

Article • 10/25/2023

The `!wdfkd.wdfextendwatchdog` extension extends the time-out period (from 10 minutes to 24 hours) of the framework's watchdog timer during power transitions.

dbgcmd

```
!wdfkd.wdfextendwatchdog Handle [Extend]
```

Parameters

Handle

A handle to a WDFDEVICE-typed object.

Extend

Optional. A value that indicates whether to enable or disable extension of the time-out period. If *Extend* is 0, extension is disabled, and the time-out period is 10 minutes. If *Extend* is 1, extension is enabled and the time-out period is 24 hours. The default value is 1.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The framework starts an internal watchdog timer every time it calls a power policy or power event callback function for a driver that is not power pageable (that is, the DO_POWER_PAGABLE bit is clear). If the callback function causes paging I/O and

therefore blocks, the operating system hangs because no paging device is available to service the request.

If the time-out period elapses, the framework issues bug check 0x10D (WDF_VIOLATION). For details, see [Bug Check 0x10D](#).

!wdfkd.wdffindobjects

Article • 10/25/2023

The **!wdfkd.wdffindobjects** extension searches memory for WDF objects.

dbgcmd

```
!wdfkd.wdffindobjects [StartAddress [Flags]]
```

Parameters

StartAddress

Optional. Specifies the address at which the search must begin. If this is omitted, the search will begin from where the most recent **!wdfkd.wdffindobjects** search ended.

Flags

Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0. *Flags* cannot be used unless *StartAddress* is specified.

Bit 0 (0x1)

Displays verbose output.

Bit 1 (0x2)

Displays internal type information for each handle.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following examples show the output of the !wdfkd.wdffindobjects extension. The 0x1 flag is set in the second example.

dbgcmd

```
1: kd> !wdffindobjects 0xfffffa600211b668
Address          Value          Object
-----
0xfffffa600211b668 0x0000000000000008
0xfffffa600211b670 0xfffffa8002e7b1f0  !WDFREQUEST 0x0000057ffd184e08
0xfffffa600211b678 0x0000000000000004
0xfffffa600211b680 0x0000000000000001
0xfffffa600211b688 0xfffffa8006aa3640  !WDFUSBPIPE 0x0000057ff955c9b8
0xfffffa600211b690 0x0000000000000000
0xfffffa600211b698 0xfffff80001e61f78
0xfffffa600211b6a0 0x0000000000000010
0xfffffa600211b6a8 0x0000000000010286
0xfffffa600211b6b0 0xfffffa600211b6c0
0xfffffa600211b6b8 0x0000000000000000
0xfffffa600211b6c0 0xfffffa8006aa3640  !WDFUSBPIPE 0x0000057ff955c9b8
0xfffffa600211b6c8 0x0000057ffd184e08  !WDFREQUEST 0x0000057ffd184e08
0xfffffa600211b6d0 0x0000000000000000
0xfffffa600211b6d8 0x0000057ffc51ea18  !WDFMEMORY 0x0000057ffc51ea18
0xfffffa600211b6e0 0x0000000000000000

1: kd> !wdffindobjects 0xfffffa600211b668 1
Address          Value          Type    Object
-----
0xfffffa600211b668 0x0000000000000008
0xfffffa600211b670 0xfffffa8002e7b1f0  Object  !WDFREQUEST
0x0000057ffd184e08
0xfffffa600211b678 0x0000000000000004
0xfffffa600211b680 0x0000000000000001
0xfffffa600211b688 0xfffffa8006aa3640  Object  !WDFUSBPIPE
0x0000057ff955c9b8
0xfffffa600211b690 0x0000000000000000
0xfffffa600211b698 0xfffff80001e61f78
0xfffffa600211b6a0 0x0000000000000010
0xfffffa600211b6a8 0x0000000000010286
0xfffffa600211b6b0 0xfffffa600211b6c0
0xfffffa600211b6b8 0x0000000000000000
0xfffffa600211b6c0 0xfffffa8006aa3640  Object  !WDFUSBPIPE
0x0000057ff955c9b8
0xfffffa600211b6c8 0x0000057ffd184e08  Handle  !WDFREQUEST
0x0000057ffd184e08
0xfffffa600211b6d0 0x0000000000000000
0xfffffa600211b6d8 0x0000057ffc51ea18  Handle  !WDFMEMORY
0x0000057ffc51ea18
0xfffffa600211b6e0 0x0000000000000000
```

!wdfkd.wdfforwardprogress

Article • 10/25/2023

The **!wdfkd.wdfforwardprogress** extension displays information about the forward progress of a specified framework queue object.

dbgcmd

```
!wdfkd.wdfforwardprogress Handle
```

Parameters

Handle

A handle to a framework queue object.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information about how to debug Kernel-Mode Driver Framework (KMDF) drivers, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

This extension will succeed only if the specified framework queue object is configured to support forward progress. If this extension is used with other objects, an error message will be displayed.

The following example shows the display from a **!wdfkd.wdfforwardprogress** extension.

dbgcmd

```
kd> !wdfkd.wdfforwardprogress 0x79af3250
```

```
# Dumping forward progress fields for WDFQUEUE 0x79af3250
=====
ForwardProgressReservedPolicy: UseExamine (0x2)

Total reserved requests: 44

Number of available reserved requests in list: 41
!WDFREQUEST 0x7bcacf4c0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc67eb0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bccf678 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bb6ce40 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7be30a58 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x79af37d0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc7f428 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bbd40f0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd333a8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd241d8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd594e0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd80d10 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x78ea2d50 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x792020f0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc37258 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bbc1fb0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bbc4fb0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7be0cb80 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc84890 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x78acbd18 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bcf1ad8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bead540 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7922c0f0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7a34a0f0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x625195d0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc33640 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bba9f28 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bba44c8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bb77cd8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7a2b89a8 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7a41ab88 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bc7cc88 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd37180 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bca40f0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x64b4af20 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd01a40 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7a25cfb0 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bba9330 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bd14a40 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7bcc0210 (Reserved) !IRP 0x00000000
!WDFREQUEST 0x7a54eb00 (Reserved) !IRP 0x00000000

Number of reserved requests in use: 3
!WDFREQUEST 0x7bf0ab80 (Reserved) !IRP 0x8438f008
!WDFREQUEST 0x7bc53ca8 (Reserved) !IRP 0x875f59f0
!WDFREQUEST 0x7bcd8b8 (Reserved) !IRP 0x85c25348

Number of undispatched IRP's in list: 0
```

EvtIoReservedResourcesAllocate: (0x9a3f1b70)
mqueue!EvtIoAllocateResourcesForReservedRequest
EvtIoExamineIrp: (0x9a3f19d0) mqueue!EvtIoWdmIrpForForwardProgress

!wdfkd.wdfgetdriver

Article • 10/25/2023

The **!wdfkd.wdfgetdriver** extension displays the name of the current default driver.

```
dbgcmd
```

```
!wdfkd.wdfgetdriver
```

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfhandle

Article • 10/25/2023

The **!wdfkd.wdfhandle** extension displays information about a specified framework object handle, such as the handle type, object context pointers, and the underlying framework object pointer.

dbgcmd

```
!wdfkd.wdfhandle Handle [Flags]
```

Parameters

Handle

A handle to a framework object.

Flags

Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 4 (0x10)

The display will include the subtree of child objects for the specified handle.

Bit 5 (0x20)

The display will include context and callback function information for the specified handle. This flag is valid only when bit 4 (0x10) is set.

Bit 6 (0x40)

The display will include additional information for the specified handle. This flag is valid only when bit 4 (0x10) is set.

Bit 7 (0x80)

The handle information will be displayed in a more compact format.

Bit 8 (0x100)

The display will left align internal type information. This flag is valid only when bit 4 (0x10) is set.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example shows the output of the `!wdfhandle` extension with bit 4 set in the *Flags* parameter (so the output displays information about the child objects).

```
dbgcmd

kd> !wdfhandle 0x7ca7b1c0 10

handle 0x7ca7b1c0, type is WDFDEVICE

Contexts:
    context: dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
    <no associated attribute callbacks>

Child WDFHANDLEs of 0x7ca7b1c0:
    WDFDEVICE 0x7ca7b1c0
        WDFCMRESLIST 0x7ccfb058
        WDFCMRESLIST 0x7cadb058
        WDFCHILDLIST 0x7c72f0c8
        WDFCHILDLIST 0x7cc090c8
        WDFIOTARGET 0x7c9630b8

!wdfobject 0x83584e38
```

In the preceding example, the input handle refers to a WDFDEVICE object. This particular device object has five child objects--two WDFCMRESLIST objects, two WDFCHILDLIST objects, and one WDFIOTARGET object.

!wdfkd.wdfhelp

Article • 10/25/2023

The **!wdfkd.wdfhelp** extension displays help information about all Wdfkd.dll extension commands.

```
dbgcmd
```

```
!wdfkd.wdfhelp
```

DLL

Wdfkd.dll

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The **!wdfkd.wdfhelp** extension is equivalent to the [**!wdfkd.help**](#) extension.

!wdfkd.wdfinterrupt

Article • 10/25/2023

The **!wdfkd.wdfinterrupt** extension displays information about a WDFINTERRUPT object.

dbgcmd

```
!wdfkd.wdfinterrupt Handle [Flags]
```

Parameters

Handle

A handle to a WDFINTERRUPT object.

Flags

Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)

Displays the interrupt service routines (ISRs) for the interrupt dispatch table (IDT) associated with this WDFINTERRUPT object. Setting this flag is equivalent to following the **!wdfinterrupt** extension with the [!idt](#) extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example shows the output of the **!wdfinterrupt** extension with bit 0 set in the *Flags* parameter (so the output displays information about the IDT).

```
dbgcmd
```

```
kd> !wdfkd.wdfinterrupt 0x7a988698 1

# Dumping WDFINTERRUPT 0x7a988698
=====
Interrupt Type: Line-based, Connected, Enabled
Vector: 0xa1 (!idt 0xa1)
Irql: 0x9
Mode: LevelSensitive
Polarity: WdfInterruptPolarityUnknown
ShareDisposition: CmResourceShareShared
FloatingSave: FALSE
Interrupt Priority Policy: WdfIrqPriorityUndefined
Processor Affinity Policy: WdfIrqPolicyOneCloseProcessor
Processor Group: 0
Processor Affinity: 0x3

dt nt!KINTERRUPT 0x8594eb28

EvtInterruptIsr: 1394ohci!Interrupt::WdfEvtInterruptIsr (0x8d580552)
EvtInterruptDpc: 1394ohci!Interrupt::WdfEvtInterruptDpc (0x8d580682)

Dumping IDT:

a1:          85167a58 ndis!ndisMiniportIsr (KINTERRUPT 85167a00)
                         Wdf01000!FxInterrupt::_InterruptThunk
(KINTERRUPT 85987500)

To get ISR from KINTERRUPT:
dt <KINTERRUPT> nt!KINTERRUPT ServiceContext
dt <ServiceContext> wdf01000!FxInterrupt m_EvtInterruptIsr
```

In the preceding example, the display concludes with two suggested [dt \(Display Type\)](#) commands that can be used to display additional data.

!wdfkd.wdfiotarget

Article • 10/25/2023

The `!wdfkd.wdfiotarget` extension displays information about a specified I/O target object.

dbgcmd

```
!wdfkd.wdfiotarget Handle [Flags]
```

Parameters

Handle

A handle to an I/O target object.

Flags

Optional. Flags that specify the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)

The display will include details for each of the I/O target's pending request objects.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example shows a display from the `!wdfkd.wdfiotarget` extension.

dbgcmd

```
kd> !wdfiotarget 0x7c9630b8

# WDFIOTARGET 8369cf40
=====
WDFDEVICE: 0x7ca7b1c0
Target Device: !devobj 0x81ede5d8
Target PDO: !devobj 0x822b8a90

Type: Instack target
State: WdfIoTargetStarted

Requests waiting: 0

Requests sent: 0

Requests sent with ignore-target-state: 0
```

The output in the preceding example includes the address of the I/O target's parent framework device object, along with the addresses of the WDM DEVICE_OBJECT structures that represent the target driver's device object and the target device's physical device object (PDO).

!wdfkd.wdfldr

Article • 10/25/2023

The **!wdfkd.wdfldr** extension displays information about the drivers that are currently dynamically bound to the Windows Driver Frameworks. This includes both the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF).

```
dbgcmd
```

```
!wdfkd.wdfldr [DriverName]
```

Parameters

DriverName

The name of a driver, including the filename extension. If you supply a driver name, this command displays detailed information about the one driver. If you do not supply a drive name, this command displays information about all drivers that are bound to the Windows Driver Frameworks.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

Here is an example of the output of **!wdfldr**.

```
dbgcmd
```

```
## 0: kd> !wdfkd.wdfldr
## KMDF Drivers
```

```

## LoadedModuleList      0xfffff800003b61f8

LIBRARY_MODULE  0xfffffe0000039f7c0
Version        v1.13
Service         \Registry\Machine\System\CurrentControlSet\Services\Wdf01000
ImageName       Wdf01000.sys
ImageAddress    0xfffff800002e7000
ImageSize       0xc5000
Associated Clients: 16

  ImageName           Ver   WdfGlobals          FxGlobals
ImageAddress     ImageSize
  peauth.sys          v1.7  0xfffffe00003a95880 0xfffffe00003a956e0
0xfffff80002678000 0x000ab000
  monitor.sys         v1.11 0xfffffe000001abc70 0xfffffe000001abad0
0xfffff800022e7000 0x0000e000
  UsbHub3.sys         v1.11 0xfffffe000028a47b0 0xfffffe000028a4610
0xfffff8000220b000 0x00077000
##   ...

## Total: 1 library loaded

## UMDF Drivers

DriverManagerProcess: 0xfffffe00003470500

  ImageName           Ver
MyUmdfDriver.dll  v1.11
SomeUmdf2Driver.dll v2.0
MyUmdf2Driver.dll  v2.0

```

Here is another example that supplies a driver name.

```

dbgcmd

0: kd> !wdflldr MyUmdf2Driver.dll

Version      v2.0
Service       \Registry\Machine\System\CurrentControlSet\Services\MyUmdf2Driver

## !wdfllogdump MyUmdf2Driver.dll

## UMDF Device Instances using MyUmdf2Driver.dll

Process      DevStack      DeviceId
0xfffffe00000c32900  a5a3ab5f70  \Device\00000052 !wdfdriverinfo

```

!wdfkd.wdflogdump

Article • 10/25/2023

The `!wdfkd.wdflogdump` extension displays the WDF In-flight Recorder log records, if available, for a KMDF driver or a UMDF 2 driver. You can use this command with a [complete memory dump](#), a [kernel memory dump](#), or a [live kernel-mode target](#).

KMDF

dbgcmd

```
!wdfkd.wdflogdump [DriverName][WdfDriverGlobals][-d | -f | -a LogAddress]
```

UMDF

dbgcmd

```
!wdfkd.wdflogdump [DriverName.dll][HostProcessId][-d | -f | -m]
```

Parameters

DriverName

- KMDF: The name of a KMDF driver. The name must not include the .sys filename extension.
- UMDF: The name of a UMDF 2 driver. The name must include the .dll filename extension.

Parameter2

- KMDF: *WdfDriverGlobals* - The address of the *WdfDriverGlobals* structure. You can determine this address by running [!wdfkd.wdfldr](#) and looking for the field labeled "WdfGlobals". Or, you can supply @@(Driver!WdfDriverGlobals) as the address value, where *Driver* is the name of the driver. If any *WdfDriverGlobals* address is supplied, *DriverName* is ignored (although it must nevertheless be supplied).
- UMDF: *HostProcessId* - The process ID of an instance of wudfhost.exe. If you supply the process ID, this command displays the log records for that process. If you do not supply the process ID, this command displays a list of commands in this form:

`!wdflogdump DriverName ** ProcessID`**

If a single process can be determined it will automatically be chosen.

Options KMDF:

-d Displays only the driver logs.

-f Displays only the framework logs.

-a *LogAddress* Displays a specific driver log. If this option is used, the LogAddress must be provided.

UMDF:

-d Displays only the driver logs.

-f Displays only the framework logs.

-m Merges framework and driver logs in their recorded order.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Remarks

If you omit the *DriverName* parameter, the default driver name is used. Use the [!wdfkd.wdfgetdriver](#) extension to display the default driver name, and use the [!wdfkd.wdfsetdriver](#) extension to set the default driver name.

To display the framework's error log records from a [small memory dump](#), use the [!wdfkd.wdfcrashdump](#) extension.

For information about setting information that the debugger needs to format WPP tracing messages, see [!wdfkd.wdftmffile](#) and [!wdfkd.wdfsettraceprefix](#).

Additional Information

For information about enabling the inflight trace recorder for your driver, see [Using Inflight Trace Recorder \(IFR\) in KMDF and UMDF 2 Drivers](#). For more information about

debugging WDF drivers, see [Debugging WDF Drivers](#). For information about KMDF debugging, see [Kernel-Mode Driver Framework Debugging](#).

See also

[!wdfkd.wdfcrashdump](#)

[!wdfkd.wdfsettraceprefix](#)

!wdfkd.wdflogsave

Article • 10/25/2023

The **!wdfkd.wdflogsave** extension saves the Kernel-Mode Driver Framework (KMDF) error log records for a specified driver to an event trace log (.etl) file that you can view by using TraceView.

dbgcmd

```
!wdfkd.wdflogsave [DriverName [FileName]]
```

Parameters

DriverName

Optional. The name of a driver. *DriverName* must not include the .sys file name extension.

FileName

Optional. The name of the file to which the KMDF error log records should be saved. *FileName* should not include the .etl file name extension. If you omit *FileName*, the KMDF error log records are saved to the *DriverName.etl* file.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

If you omit the *DriverName* parameter, the default driver name is used. Use the [!wdfkd.wdfgetdriver](#) extension to display the default driver name, and use the [!wdfkd.wdfsetdriver](#) extension to set the default driver name.

!wdfkd.wdfmemory

Article • 10/25/2023

The **!wdfkd.wdfmemory** extension displays the address and size of the buffer that is associated with a framework memory object.

dbgcmd

```
!wdfkd.wdfmemory Handle
```

Parameters

Handle

A handle to a framework memory object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfobject

Article • 10/25/2023

The **!wdfkd.wdfobject** extension displays information about a specified framework object.

```
dbgcmd
```

```
!wdfkd.wdfobject FrameworkObject
```

Parameters

FrameworkObject

A pointer to a framework object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

If the Kernel-Mode Driver Framework (KMDF) verifier is enabled for a driver and the public handle type was marked for tracking, the display from the **!wdfkd.wdfobject** extension includes the tag tracker (that is, the tracking object), as in the following example.

```
dbgcmd
```

```
kd> !wdfobject 0x83584e38
```

```
The type for object 0x83584e38 is FxDevice
State: FxObjectStateCreated (0x1)
```

```
!wdfhandle 0x7ca7b1c0

dt FxDevice 0x83584e38

    context: dt 0x83584ff8 ROOT_CONTEXT (size is 0x1 bytes)
        <no associated attribute callbacks>

Object debug extension 83584e20
    !wdftagtracker 0x83722d80
    Verifier lock 0x831cefa8

State history:
[0] FxObjectStateChanged (0x1)
```

!wdfkd.wdfopenhandles

Article • 10/25/2023

The `!wdfkd.wdfopenhandles` extension displays information about all the handles that are open on the specified WDF device.

dbgcmd

```
!wdfkd.wdfopenhandles Handle [Flags]
```

Parameters

Handle

A handle to a framework device object (WDFDEVICE).

Flags

Optional. Specifies the kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)

Displays file object context information.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfpoolusage

Article • 10/25/2023

The **!wdfkd.wdfpoolusage** extension displays pool usage information for a specified driver, if the Kernel-Mode Driver Framework (KMDF) verifier is enabled for the driver.

dbgcmd

```
!wdfkd.wdfpoolusage [DriverName [SearchAddress] [Flags]]
```

Parameters

DriverName

Optional. The name of a driver. *DriverName* must not include the .sys file name extension.

SearchAddress

Optional. A string that represents a memory address. The pool entry that contains *SearchAddress* is displayed. If *SearchAddress* is 0 or omitted, all of the driver's pool entries are displayed.

Flags

Optional. The kind of information to display. This parameter is valid only if *SearchAddress* is nonzero. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)

Displays verbose output. Multiple lines are displayed for each. If this flag is not set, the information about an allocation is displayed on one line.

Bit 1 (0x2)

Displays internal type information for each handle.

Bit 2 (0x4)

Displays the caller of each pool entry.

DLL

Wdfkd.dll

Frameworks

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

If you omit the *DriverName* parameter, the default driver is used. You can display the default driver by using the **!wdfkd.wdfgetdriver** extension; you can set the default driver by using the **!wdfkd.wdfsetdriver** extension.

The following example shows the output from the **!wdfpoolusage** extension when no pool allocation is marked and the *Flags* value is set to 0.

```
dbgcmd

## kd> !wdfpoolusage wdfrawbusenumtest 0 0
-----
## FxDriverGlobals 83b7af18 pool stats
-----
Driver Tag: 'RawB'
15126 NonPaged Bytes, 548 Paged Bytes
94 NonPaged Allocations, 10 Paged Allocations
15610 PeakNonPaged Bytes, 752 PeakPaged Bytes
100 PeakNonPaged Allocations, 14 PeakPaged Allocations

pool 82dbae00, Size 512 Tag 'RawB', NonPaged, Caller:
Wdf01000!FxVerifierLock::AllocateThreadTable+5d
```

The following example shows the output from **!wdfpoolusage** that appears when the value of *Flags* is 1. (Note that the ellipsis (...) on the second line indicates the omission of some output that is the same as that shown in the preceding example.)

```
dbgcmd

kd> !wdfpoolusage wdfrawbusenumtest 0 1
...
100 PeakNonPaged Allocations, 14 PeakPaged Allocations

Client alloc starts at 82dbae00
Size 512 Tag 'RawB'
NonPaged (0x0)
Caller: Wdf01000!FxVerifierLock::AllocateThreadTable+5d
```

!wdfkd.wdfqueue

Article • 10/25/2023

The **!wdfkd.wdfqueue** extension displays information about a specified framework queue object and the framework request objects that are in the queue.

```
dbgcmd
```

```
!wdfkd.wdfqueue Handle
```

Parameters

Handle

A handle to a framework queue object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The following example shows the display from a **!wdfkd.wdfqueue** extension.

```
dbgcmd
```

```
kd> !wdfqueue 0x7ce7d1e8
# Dumping WDFQUEUE 0x7ce7d1e8
=====
Parallel, Power-managed, PowerOff, Can accept, Can dispatch,
ExecutionLevelDispatch, SynchronizationScopeNone
    Number of driver owned requests: 0
    Number of waiting requests: 0
```

```
EvtIoDefault: (0xf221fad0) wdfrawbusenumtest!EvtIoQueueDefault
```

The queue in the preceding example is configured for parallel dispatching, is power-managed but is currently in the Off state, and can both accept and dispatch requests.

!wdfkd.wdfrequest

Article • 10/25/2023

The **!wdfkd.wdfrequest** extension displays information about a specified framework request object and the WDM I/O request packet (IRP) that is associated with the request object.

dbgcmd

```
!wdfkd.wdfrequest Handle
```

Parameters

Handle

A handle to a framework request object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfsearchpath

Article • 04/03/2024

The `!wdfkd.wdfsearchpath` extension sets the search path to formatting files for Kernel-Mode Driver Framework (KMDF) error log records.

dbgcmd

`!wdfkd.wdfsearchpath Path`

Parameters

Path

The path of a directory that contains KMDF formatting files.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The KMDF formatting files are included in the Windows Driver Kit (WDK). The path to the formatting files depends on the installation directory of your WDK and on the version of the WDK that you have installed. The KMDF formatting files have extension tmf (trace message formatting). To determine the search path, browse or search your WDK installation for file names of the form `WdfVersionNumber.tmf`. The following example shows how to use the `!wdfkd.wdfsearchpath` extension.

dbgcmd

```
kd> !wdfsearchpath C:\WinDDK\7600\tools\tracing\amd64
```

The TRACE_FORMAT_SEARCH_PATH environment variable also controls the search path, but the **!wdfkd.wdfsearchpath** extension takes precedence over the search path that TRACE_FORMAT_SEARCH_PATH specifies.

Feedback

Was this page helpful?

 Yes

 No

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

!wdfkd.wdfsettraceprefix

Article • 10/25/2023

The !wdfkd.wdfsettraceprefix extension sets the trace prefix format string.

dbgcmd

!wdfkd.wdfsettraceprefix String

Parameters

String

A trace prefix string.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The trace prefix string is prepended to each trace message in the Kernel-Mode Driver Framework (KMDF) error log. The TRACE_FORMAT_PREFIX environment variable also controls the trace prefix string.

The format of the trace prefix string is defined by the Microsoft Windows tracing tools. For more information about the format of this string and how to customize it, see the "Trace Message Prefix" topic in the Driver Development Tools section of the Windows Driver Kit (WDK).

!wdfkd.wdfsetdriver

Article • 10/25/2023

The **!wdfkd.wdfsetdriver** extension sets the name of the default Kernel-Mode Driver Framework (KMDF) driver to which debugger extension commands apply.

dbgcmd

```
!wdfkd.wdfsetdriver DriverName
```

Parameters

DriverName

The name of a driver. *DriverName* must not include the .sys file name extension.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The **!wdfkd.wdfsetdriver** extension sets the default driver name. You can use this name with other **wdfkd** extensions that would otherwise require you to specify a driver name.

To obtain the name of the current default KMDF driver, use the [!wdfkd.wdfgetdriver](#) extension.

!wdfkd.wdfspinlock

Article • 10/25/2023

The **!wdfkd.wdfspinlock** extension displays information about a framework spin-lock object. This information includes the spin lock's acquisition history and the length of time that the lock was held.

dbgcmd

```
!wdfkd.wdfspinlock Handle
```

Parameters

Handle

A handle to a WDFSPINLOCK-typed object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdftagtracker

Article • 10/25/2023

The **!wdfkd.wdftagtracker** extension displays all available tag information (including tag value, line, file, and time) for a specified tag tracker.

dbgcmd

```
!wdfkd.wdftagtracker TagObjectPointer [Flags]
```

Parameters

TagObjectPointer

A pointer to a tag tracker.

Flags

Optional. The kind of information to display. *Flags* can be any combination of the following bits. The default value is 0x0.

Bit 0 (0x1)

Displays the history of acquire operations and release operations on the object.

Bit 1 (0x2)

Displays the line number of the object in hexadecimal instead of decimal.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

To retrieve a pointer to a tag tracker, use the **!wdfkd.wdfobject** extension on an internal framework object pointer.

To use tag tracking, you must enable both the Kernel-Mode Driver Framework (KMDF) verifier and handle tracking in the registry. Both of these settings are stored in the driver's **Parameters\Wdf** subkey of the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services** key.

To enable the KMDF verifier, set a nonzero value for **VerifierOn**.

To enable handle tracking, set the value of **TrackHandles** to the name of one or more object types, or specify an asterisk (*) to track all object types. For example, the following example specifies the tracking of references to all WDFDEVICE and WDFQUEUE objects.

text

```
TrackHandles: MULTI_SZ: WDFDEVICE WDFQUEUE
```

When you enable handle tracking for an object type, the framework tracks the references that are taken on any object of that type. This setting is useful in finding driver memory leaks that unreleased references cause. **TrackHandles** works only if the KMDF verifier is enabled.

!wdfkd.wdftmffile

Article • 10/25/2023

The **!wdfkd.wdftmffile** extension sets the trace message format (.tmf) file to use when the debugger is formatting Kernel-Mode Driver Framework (KMDF) error log records for the [!wdfkd.wdflogdump](#) or [!wdfkd.wdfcrashdump](#) extensions.

dbgcmd

```
!wdfkd.wdftmffile TMFpath
```

Parameters

TMFpath

A path that contains the .tmf file.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

If your driver uses a KMDF version earlier than 1.11, you must use the **!wdfkd.wdftmffile** extension before you can use the [!wdfkd.wdflogdump](#) or [!wdfkd.wdfcrashdump](#) extensions.

Starting in KMDF version 1.11, the framework library's symbol file (for example wdf01000.pdb) contains the trace message format (TMF) entries. Starting in the Windows 8 version of the kernel debugger, the [Kernel-Mode Driver Framework Extensions \(Wdfkd.dll\)](#) read the entries from the .pdb file. As a result, if your driver uses KMDF version 1.11 or later, and you are using the kernel debugger from Windows 8 or

later, you do not need to use `!wdfkd.wdftmffile`. You do need to include the directory that contains the symbol file in the debugger's [symbol path](#). The debugging target machine can be running any operating system that supports KMDF.

The following example shows how to use the `!wdfkd.wdftmffile` extension from the root WDK directory, for KMDF version 1.5.

```
dbgcmd
```

```
kd> !wdftmffile tools\tracing\<platform>\wdf1005.tmf
```

Note that the path might be different for the version of the Windows Driver Kit (WDK) that you are using. Also note that the .tmf file's name represents the version of KMDF that you are using. For example, Wdf1005.tmf is the .tmf file for KMDF version 1.5.

For information about how to view the KMDF log during a debugging session, see [Using the Framework's Event Logger](#).

!wdfkd.wdftraceprtdebug

Article • 10/25/2023

The **!wdfkd.wdftraceprtdebug** extension enables and disables the Traceprt.dll diagnostic mode, which generates verbose debugging information.

dbgcmd

```
!wdfkd.wdftraceprtdebug {on | off}
```

Parameters

on

Enables the Traceprt.dll diagnostic mode.

off

Disables the Traceprt.dll diagnostic mode.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You should use the **!wdfkd.wdftraceprtdebug** extension only at the direction of technical support.

!wdfkd.wdfumdevstack

Article • 10/25/2023

The **!wdfkd.wdfumdevstack** extension displays detailed information about a UMDF device stack in the [implicit process](#).

dbgcmd

```
!wdfkd.wdfumdevstack DevstackAddress [Flags]
```

Parameters

DevstackAddress

Specifies the address of the device stack to display information about. You can use [!wdfkd.wdfumdevstacks](#) to get the addresses of UMDF device stacks in the implicit process.

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays detailed information about the device stack.

Bit 7 (0x80)

Displays information about the internal framework.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command [!wdfext.umdevstack](#).

Here is an example of how to use [!wdfumdevstack](#). First use [!wdfumdevstacks](#) to display the UMDF device stacks in the implicit process.

```
dbgcmd

0: kd> !wdfkd.wdfumdevstacks
Number of device stacks: 1
Device Stack: 0x000000a5a3ab5f70      Pdo Name: \Device\00000052
  Active: Yes
  Number of UM devices: 1
  Device 0
    Driver Config Registry Path: MyUmdf2Driver
    UMDriver Image Path:
C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
  FxDriver: 0xa5a3acaaa0
  FxDevice: 0xa5a3ac4fc0
  Open UM files (use !wdfumfile <addr> for details): <None>
  Device XferMode: Deferred RW: Buffered CTL: Buffered
  DevStack XferMode: Deferred RW: Buffered CTL: Buffered
```

The preceding output shows that there is one UMDF device stack in the implicit process. You can also see that the device stack has one device object (Number of UM devices: 1).

The preceding output displays the address of a device stack (0x000000a5a3ab5f70). To get detailed information about the device stack, pass its address to [!wdfumdevstack](#). In this example, we set the *Flags* parameter to 0x80 to include information about the framework.

```
dbgcmd

0: kd> !wdfkd.wdfumdevstack 0x000000a5a3ab5f70 0x80
Device Stack: 0x000000a5a3ab5f70      Pdo Name: \Device\00000052
  Active: Yes
  Number of UM devices: 1
  Device 0
    Driver Config Registry Path: MyUmdf2Driver
    UMDriver Image Path:
C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
  FxDriver: 0xa5a3acaaa0
  FxDevice: 0xa5a3ac4fc0
  Open UM files (use !wdfumfile <addr> for details): <None>
```

```
Device XFERMODE: Deferred RW: Buffered CTL: Buffered
Internal Values:
    wudfhost!WudfDriverAndFxInfo 0x000000a5a3ac21b8
    IUMDFramework: 0x0000000000000000
    IFxMessageDispatch: 0x000000a5a3aba630
    FxDevice 0x000000a5a3ac4fc0
    Modules:
        Driver: wudfhost!CWudfModuleInfo 0x000000a5a3ac18f0
        Fx:      wudfhost!CWudfModuleInfo 0x000000a5a3aca7a0
        wudfx02000!FxDriver: 0x000000a5a3acaaa0
DevStack XFERMODE: Deferred RW: Buffered CTL: Buffered
```

!wdfkd.wdfumdevstacks

Article • 10/25/2023

The `!wdfkd.wdfumdevstacks` extension displays information about all UMDF device stacks in the [implicit process](#).

dbgcmd

```
!wdfkd.wdfumdevstacks [Flags]
```

Parameters

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays detailed information about each device stack.

Bit 7 (0x80)

Displays information about the internal framework.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (`wudfhost.exe`).

This command displays the same information as the user-mode command [!wudfext.umdevstacks](#).

Before you use this command, use [!process](#) to get a list of all UMDF host processes.

```
dbgcmd

0: kd> !process 0 0 wudfhost.exe
PROCESS fffffe00000c32900
    SessionId: 0 Cid: 079c Peb: 7ff782537000 ParentCid: 037c
    DirBase: 607af000 ObjectTable: fffffc00009807940 HandleCount: <Data Not
Accessible>
    Image: WUDFHost.exe
```

The preceding output shows that there is one UMDF host process; that is, there is one instance of wudfhost.exe.

Next use [.process](#) to set the implicit process to wudfhost.exe.

```
dbgcmd

0: kd> .process /P fffffe00000c32900
Implicit process is now fffffe000`00c32900
.cache forcedecodeptes done
```

Now use [!wdfkd.wdfumdevstacks](#) to display the UMDF device stacks in the implicit process (wudfhost.exe).

```
dbgcmd

0: kd> !wdfkd.wdfumdevstacks
Number of device stacks: 1
    Device Stack: 0x000000a5a3ab5f70 Pdo Name: \Device\00000052
        Active: Yes
        Number of UM devices: 1
        Device 0
            Driver Config Registry Path: MyUmdf2Driver
            UMDriver Image Path:
            C:\WINDOWS\System32\drivers\UMDF\MyUmdf2Driver.dll
            FxDriver: 0xa5a3acaaa0
            FxDevice: 0xa5a3ac4fc0
            Open UM files (use !wdfumfile <addr> for details): <None>
            Device XferMode: Deferred RW: Buffered CTL: Buffered
            DevStack XferMode: Deferred RW: Buffered CTL: Buffered
```

The preceding output shows that there is one UMDF device stack in the implicit process. You can also see that the device stack has one device object (Number of UM devices: 1).

!wdfkd.wdfumdownirp

Article • 10/25/2023

The `!wdfkd.wdfumdownirp` extension displays the kernel-mode I/O request packet (IRP) that is associated with a specified user-mode IRP. This command is used in two steps.

See Remarks.

dbgcmd

```
!wdfkd.wdfumdownirp UmIrp [FileObject]
```

Parameters

UmIrp

Specifies the address of a user mode IRP. You can use [!wdfkd.wdfumirps](#) to get the addresses of UM IRPs in the [implicit process](#).

FileObject

Specifies the address of a `_FILE_OBJECT` structure. For information about how to get this address, see Remarks.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (`wudfhost.exe`).

To use this command, follow these steps:

1. Enter this command, passing only the address a user-mode IRP. The command displays a handle.
2. Pass the displayed handle to the **!handle** command. In the output of **!handle**, find the address of a **_FILE_OBJECT** structure.
3. Enter this command again, passing both the address of the user-mode IRP and the address of the **_FILE_OBJECT** structure.

!wdfkd.wdfumfile

Article • 10/25/2023

The **!wdfkd.wdfumfile** extension displays information about a UMDF intra-stack file.

dbgcmd

!wdfkd.wdfumfile Address

Parameters

Address

Specifies the address of the UMDF intra-stack file to display information about.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command [**!wudfext.umfile**](#).

!wdfkd.wdfumirp

Article • 10/25/2023

The **!wdfkd.wdfumirp** extension displays information about a user-mode I/O request packet (UM IRP).

dbgcmd

!wdfkd.wdfumirp Address

Parameters

Address

Specifies the address of the UM IRP to display information about. You can use [!wdfkd.wdfumirps](#) to get the addresses of UM IRPs in the [implicit process](#).

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command [!wudfext.umirp](#).

You can use [!process](#) to get a list of all UMDF host processes, and you can use [.process](#) to set the implicit process to one of the UMDF host processes. For a detailed example, see [!wdfkd.wdfumdevstacks](#).

The following shows how to use **!wdfkd.wdfumirps** and **!wdfkd.wdfumirp** to display information about an individual UM IRP.

```
dbgcmd

0: kd> !wdfkd.wdfumirps
Number of pending IRPS: 0x4
#####  CWudfIrp      Current Type          UniqueId KernelIrp      Device
Stack
-----
-----
...
0003  1ab9eae370  Power (WAIT_WAKE)      0      fffffe00000c53010
1ab9eaa6d0

0: kd> !wdfkd.wdfumirp 1ab9eae370
UM IRP: 0x0000001ab9eae370  UniqueId: 0x0  Kernel Irp: 0xfffffe00000c53010
  Type: Power (WAIT_WAKE)
  ClientProcessId: 0x0
  Device Stack: 0x0000001ab9eaa6d0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Total number of stack locations: 2
  CurrentStackLocation: StackLocation[ 0 ]
  > StackLocation[ 0 ]
    FxDevice: (None)
    Completion:
      Callback: 0x0000000000000000
      Context: 0x0000001ab9ebc750
  StackLocation[ 1 ]
  ...

```

!wdfkd.wdfumirps

Article • 10/25/2023

The **!wdfkd.wdfumirps** extension displays the list of pending user-mode I/O request packets (UM IRPs) in the [implicit process](#).

dbgcmd

```
!wdfkd.wdfumirps NumberOfIrps Flags
```

Parameters

NumberOfIrps

Optional. Specifies the number of pending UM IRPs to display information about. If *NumberOfIrps* is an asterisk (*) or is omitted, all UM IRPs will be displayed.

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays details about the pending IRPs.

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session or in a user-mode debugging session that is attached to the UMDF host process (wudfhost.exe).

This command displays the same information as the user-mode command [!wudfext.umirps](#).

You can use [!process](#) to get a list of all UMDF host processes, and you can use [.process](#) to set the implicit process to one of the UMDF host processes. For a detailed example, see [!wdfkd.wdfumdevstacks](#).

Here is an example of the output of [!wdfkd.wdfumirps](#).

```
dbgcmd

0: kd> !wdfkd.wdfumirps
Number of pending IRPS: 0x4
#####  CWudfIrp      Current Type          UniqueId KernelIrp      Device
Stack
-----
-----
0000 1ab9e90c40  WdfRequestUndefined    0      0
1ab9eaa6d0
0001 1ab9ebfa90  WdfRequestInternalIoctl 0      0
1ab9eaa6d0
0002 1ab9ebfd10  WdfRequestInternalIoctl 0      0
1ab9eaa6d0
0003 1ab9eae370  Power (WAIT_WAKE)      0      fffffe00000c53010
1ab9eaa6d0
```

!wdfkd.wdfumtriage

Article • 10/25/2023

The **!wdfkd.wdfumtriage** extension displays information about all UMDF devices on the system, including device objects, corresponding host process, loaded drivers and class extensions, PnP device stack, PnP device nodes, dispatched IRPs, and problem state if relevant.

```
dbgcmd
```

```
!wdfkd.wdfumtriage
```

DLL

Wdfkd.dll

Frameworks

UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

You can use this command in a kernel-mode debugging session.

Here is an example of the output of **!wdfkd.wdfumtriage**.

```

0: kd> !wdfkd.wdfumtriage
WudfRd Driver Object Info
dt WudfRd!RdDriver 0xffffffff9a3e9ee8
Driver manager process 0x9a291680(Switch Context) (Dump Process)
HostFailKdDebugBreak 1

-----
UMDF FDO Info
dt WudfRd!RdFdoDevice 0xffffffff9a3ce9c8
dt WudfRd!RdProcess 0xffffffffa5863828
Device instance path Root\umdf2\FusionV2
!devnode 0xffffffff8aecfa600
!devstack 0xffffffff8aecfa800
# of UMDF drivers 2
1)SensorsCx.dll
2)FusionV2.dll

-----
UMDF FDO Info
dt WudfRd!RdFdoDevice 0xffffffffa5a26758
dt WudfRd!RdProcess 0xffffffffa5863828
Device instance path ACPI\QCOM2495\2&daba3ff&0
!devnode 0xffffffff8ee50e30
!devstack 0xffffffff8ee43a00
# of UMDF drivers 2
1)SensorsCx.dll
2)qcSensors8626.dll

-----
UMDF FDO Info
dt WudfRd!RdFdoDevice 0xffffffffa6be4980
dt WudfRd!RdProcess 0xffffffffa5863828
Device instance path SVD\umdf2\SDOV2
!devnode 0xffffffffa6be8980
!devstack 0xffffffff9a297aa0
# of UMDF drivers 2
1)SensorsCx.dll
2)SdoV2.dll

-----
Driver host process Info
Host Process 0x9a3fdb80(Switch Context) (Dump Process)
dt WudfRd!RdProcess 0xffffffffa5863828
Process timeout 60 seconds

I/O Port
WudfPf!WdfLpcCommPort a583bc30
# of pending message with timeout 0
Pending Messages:
    WUDPFf!WudfLpcMessage 9a3ec750      lirp 0xffffffff9a27008      (03/00)      - id 00000000000000ae

I/O Cancel Port
WudfPf!WdfLpcCommPort 9a3f1008
# of pending message with timeout 0

System Event Port
WudfPf!WdfLpcCommPort a585af78
# of pending message with timeout 0

Non-state changing events port
WudfPf!WdfLpcCommPort a585a498
# of pending message with timeout 0

```

!wdfkd.wdfusbdevice

Article • 10/25/2023

The !wdfkd.wdfusbdevice extension displays information about a specified Kernel-Mode Driver Framework (KMDF) USB device object. This information includes all USB interfaces and the pipes that are configured for each interface.

dbgcmd

```
!wdfkd.wdfusbdevice Handle [Flags]
```

Parameters

Handle

A handle to a WDFUSBDEVICE-typed USB device object.

Flags

Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)

The display will include the properties of the I/O target.

Bit 1 (0x2)

The display will include the properties of the I/O target for each USB pipe object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfusbinterface

Article • 10/25/2023

The **!wdfkd.wdfusbinterface** extension displays information about a specified Kernel-Mode Driver Framework (KMDF) USB interface object, including its possible and current settings.

dbgcmd

```
!wdfkd.wdfusbinterface Handle [Flags]
```

Parameters

Handle

A handle to a WDFUSBINTERFACE-typed USB interface object.

Flags

Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)

The display will include the properties of the I/O target for each KMDF USB pipe object.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfusbpipe

Article • 10/25/2023

The `!wdfkd.wdfusbpipe` extension displays information about a Kernel-Mode Driver Framework (KMDF) USB pipe object's I/O target.

dbgcmd

```
!wdfkd.wdfusbpipe Handle [Flags]
```

Parameters

Handle

A handle to a WDFUSBPIPE-typed USB pipe object.

Flags

Optional. A hexadecimal value that modifies the kind of information to return. The default value is 0x0. Flags can be any combination of the following bits:

Bit 0 (0x1)

The display will include the properties of the I/O target.

DLL

Wdfkd.dll

Frameworks

KMDF 1, UMDF 2

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

!wdfkd.wdfwmi

Article • 10/25/2023

The **!wdfkd.wdfwmi** extension displays the Microsoft Windows Management Instrumentation (WMI) information for a specified framework device object. This information includes all WMI provider objects and their associated WMI instance objects.

```
dbgcmd
```

```
!wdfkd.wdfwmi Handle
```

Parameters

Handle

A handle to a framework device object.

DLL

Wdfkd.dll

Frameworks

KMDF 1

Additional Information

For more information, see [Kernel-Mode Driver Framework Debugging](#).

Remarks

The output of the **!wdfkd.wdfwmi** extension includes information about the WMI registration, provider, and instances.

User-Mode Driver Framework Extensions (Wudfext.dll)

Article • 10/25/2023

Extension commands that are useful for debugging User-Mode Driver Framework drivers are implemented in Wudfext.dll.

Some extensions have additional restrictions on the Windows version or UMDF version that is required; these restrictions are noted on the individual reference pages.

Note When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h. If your driver name exceeds the maximum length, your driver will fail to load.

For ways to use these extensions, see [User-Mode Driver Framework Debugging](#).

!wudfext.help

Article • 04/03/2024

The **!wudfext.help** extension displays all Wudfext.dll extension commands.

```
dbgcmd
```

```
!wudfext.help
```

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.umdevstack

Article • 04/03/2024

The **!wudfext.umdevstack** extension displays detailed information about a device stack in the host process.

dbgcmd

```
!wudfext.umdevstack DevstackAddress [Flags]
```

Parameters

DevstackAddress

Specifies the address of the device stack to display information about.

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays detailed information about the device stack.

Bit 8 (0x80)

Displays information about the internal framework.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The following is an example of the **!wudfext.umdevstack** display:

dbgcmd

```
kd> !umdevstack 0x0034e4e0
Device Stack: 0x0034e4e0 Pdo Name: \Device\00000057
Number of UM drivers: 0x1
Driver 00:
    Driver Config Registry Path: WUDFEchoDriver
    UMDriver Image Path: C:\Windows\system32\DRIVERS\UMDF\WUDFEchoDriver.dll
    Fx Driver: IWDFDriver 0xf2db8
    Fx Device: IWDFDevice 0xf2f80
    IDriverEntry: WUDFEchoDriver!CMyDriver 0x000f2c70
```

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.umdevstacks

Article • 10/25/2023

The **!wudfext.umdevstacks** extension displays information about all device stacks in the current host process.

dbgcmd

```
!wudfext.umdevstacks [Flags]
```

Parameters

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays detailed information about each device stack.

Bit 8 (0x80)

Displays information about the internal framework.

DLL

Wudfext.dll

Remarks

The **!wudfext.umdevstacks** extension displays the framework interface objects that are associated with each device stack. For more information about using the output from **!wudfext.umdevstacks**, see [!wudfext.umdevstack](#).

The **!wudfext.umdevstacks** output includes two fields entitled "Object Tracking" and "RefCount Tracking". These indicate whether the object tracking option (**TrackObjects**) and the reference count tracking option (**TrackRefCounts**) have been enabled in WDF Verifier, respectively. If the object tracking option has been enabled, the display includes the object tracker address; this address can be passed to [!wudfext.wudfdumpobjects](#) to display tracking information.

Here is an example of the **!wudfext.umdevstacks** display:

```
dbgcmd
```

```
0: kd> !umdevstacks
Number of device stacks: 1
Device Stack: 0x038c6f08      Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
    Device 0
        Driver Config Registry Path: WUDFOsrUsbFx2
        UMDriver Image Path:
D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
    Fx Driver: IWDFDriver 0x3076ff0
    Fx Device: IWDFDevice 0x3082e70
        IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0306eff8
    Open UM files (use !umfile <addr> for details):
        0x04a8ef84
    Device XferMode: CopyImmediately RW: Buffered CTL: Buffered
    Object Tracker Address: 0x03074fd8
        Object Tracking ON
        Refcount Tracking OFF
    DevStack XferMode: CopyImmediately RW: Buffered CTL: Buffered
```

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

!wudfext.umfile

Article • 04/03/2024

The **!wudfext.umfile** extension displays information about a UMDF intra-stack file.



Parameters

Address

Specifies the address of the UMDF intra-stack file to display information about.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

Yes

No

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

!wudfext.umirp

Article • 04/03/2024

The **!wudfext.umirp** extension displays information about a host user-mode I/O request packet (UM IRP).

dbgcmd

!wudfext.umirp Address

Parameters

Address

Specifies the address of the UM IRP to display information about.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

You can use the [**!wudfext.umirps**](#) extension command to display a list of all outstanding UM IRPs in the host process.

Each UM IRP has one or more stack locations. Each stack location corresponds to the parameters that a single driver in the device stack will receive when it is called to handle a request.

!wudfext.umirp dumps all of the stack locations and marks the current location with a right angle bracket (>). The current location corresponds to the driver that currently owns the request. The current location changes when a driver forwards a request to the next lower driver in the stack, or when the driver completes a request that the driver owns.

The following is an example of the **!wudfext.umirp** display:

dbgcmd

```
kd> !umirp 3dd480
UM IRP: 0x003dd480 UniqueId: 0xde Kernel Irp: 0x0x85377850
  Type: WudfMsg_READ
  ClientProcessId: 0x338
  Device Stack: 0x0034e4e0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Driver/Framework created IRP: No
  Data Buffer: 0x00000000 / 0
  IsFrom32BitProcess: Yes
  CancelFlagSet: No
  Cancel callback: 0x01102224
  Total number of stack locations: 2
  CurrentStackLocation: 2 (StackLocation[ 1 ])
    StackLocation[ 0 ]
      UNINITIALIZED
  > StackLocation[ 1 ]
    IWDFRequest: ???
    IWDFDevice: 0x000f2f80
    IWDFFile: 0x003a7648
    Completion:
      Callback: 0x00000000
      Context: 0x00000000
    Parameters: (RequestType: WdfRequestRead)
      Buffer length: 0x400
      Key: 0x00000000
      Offset: 0x0
```

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.umirps

Article • 04/03/2024

The **!wudfext.umirps** extension displays the list of pending user-mode I/O request packets (UM IRPs) in the host process.

dbgcmd

```
!wudfext.umirps NumberOfIrps Flags
```

Parameters

NumberOfIrps

Optional. Specifies the number of pending UM IRPs to display information about. If *NumberOfIrps* is an asterisk (*) or is omitted, all UM all UM IRPs will be displayed.

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Displays details about the pending IRPs.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The list of pending UM IRPs that are displayed have either been presented to the driver or are waiting to be presented to the driver.

By default, **!wudfext.umirps** shows all UM IRPs. However, you can use the *NumberOfIrps* parameter to limit this display.

The following is an example of the **!wudfext.umirps** display:

dbgcmd

####	CWudfIrp	Type	UniqueId	KernelIrp
0000	3dd280	READ	dc	856f02f0
0001	3dd380	WRITE	dd	85b869e0
0002	3dd480	READ	de	85377850
0003	3dd580	READ	df	93bba4e8
0004	3dd680	WRITE	e0	84cb9d70
0005	3dd780	READ	e1	85bec150
0006	3dd880	WRITE	e2	86651db0
0007	3dd980	READ	e3	85c22818
0008	3dda80	READ	e4	9961d150
0009	3ddb80	WRITE	e5	85c15148

To determine the corresponding kernel-mode IRP, use the [!wudfext.wudfdownkmirp](#) extension. Alternatively, you can use the values in the **UniqueId** and **KernelIrp** columns to match a UMDF IRP (or UM IRP) to a corresponding kernel IRP. You can pass the values in the **CWudfIrp** column to the [!wudfext.umirp](#) extension to determine the framework **IWDFRequest** objects that each layer in the device stack can access.

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfdevice

Article • 04/03/2024

The **!wudfext.wudfdevice** extension displays the Plug and Play (PnP) and power-management state systems for a device.

```
dbgcmd
```

```
!wudfext.wudfdevice pWDFDevice
```

Parameters

pWDFDevice

Specifies the address of the **IWDFDevice** interface to display PnP or power-management state about.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

You can use the **!wudfext.wudfdevice** extension to determine the current PnP or power-management state of the device that the *pWDFDevice* parameter specifies.

The following is an example of the **!wudfext.wudfdevice** display:

```
dbgcmd
```

```
kd> !wudfdevice 0xf2f80
Pnp Driver Callbacks:
  IPnpCallback: 0x0
  IPnpCallbackHardware: 0x0
  IPnpSelfManagedIo: 0x0
Pnp State Machine:
  CurrentState: WdfDevStatePnpStarted
  Pending UMIrp: 0x00000000
```

```
Could not read event queue depth, assuming 8
Event queue:
Processed/in-process events:
    PnpEventAddDevice
    PnpEventStartDevice
    PnpEventPwrPolStarted
Pending events:
State History:
    WdfDevStatePnpInit
    WdfDevStatePnpInitStarting
    WdfDevStatePnpHardwareAvailable
    WdfDevStatePnpEnableInterfaces
    WdfDevStatePnpStarted
Power State Machine:
    CurrentState:          WdfDevStatePowerD0
    Pending UMIrp:         0x00000000
    IoCallbackFailure:    false
    Could not read event queue depth, assuming 8
Event queue:
Processed/in-process events:
    PowerImplicitD0
Pending events:
State History:
    WdfDevStatePowerStartingCheckDeviceType
    WdfDevStatePowerD0Starting
    WdfDevStatePowerD0StartingConnectInterrupt
    WdfDevStatePowerD0StartingDmaEnable
    WdfDevStatePowerD0StartingStartSelfManagedIo
    WdfDevStatePowerDecideD0State
    WdfDevStatePowerD0
Power Policy State Machine:
    CurrentState           : WdfDevStatePwrPolStartingSucceeded
    PowerPolicyOwner       : false
    PendingSystemPower UMIrp : 0x00000000
    PowerFailed            : false
    Could not read event queue depth, assuming 8
Event queue:
Processed/in-process events:
    PwrPolStart
    PwrPolPowerUp
Pending events:
State History:
    WdfDevStatePwrPolStarting
    WdfDevStatePwrPolStarted
    WdfDevStatePwrPolStartingSucceeded
```

Feedback

Was this page helpful?

 Yes

 No

!wudfext.wudfdevicequeues

Article • 04/03/2024

The **!wudfext.wudfdevicequeues** extension displays information about all the I/O queues for a device.

```
dbgcmd
```

```
!wudfext.wudfdevicequeues pWDFDevice
```

Parameters

pWDFDevice

Specifies the address of the **IWDFDevice** interface for which to display information about all of its associated I/O queues. The **!wudfext.wudfdriverinfo** extension command determines the address of **IWDFDevice**.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The following is an example of the **!wudfext.wudfdevicequeues** display:

```
dbgcmd
```

```
## kd> !wudfdevicequeues 0xf2f80
-----
Queue: 1 (!wudfqueue 0x000f3500)
    WdfIoQueueDispatchSequential, POWER MANAGED, WdfIoQueuePowerOn, CAN
    ACCEPT, CAN DISPATCH
    Number of driver owned requests: 1
        IWDFIoRequest 0x000fa7c0      CWdfIoRequest 0x000fa748
    Number of waiting requests: 199
        IWDFIoRequest 0x000fa908      CWdfIoRequest 0x000fa890
        IWDFIoRequest 0x000faa50      CWdfIoRequest 0x000fa9d8
```

...
IWDFIoRequest 0x000fa678 CwdfIoRequest 0x000fa600
Driver's callback interfaces.
IQueueCallbackRead 0x000f343c
IQueueCallbackDeviceIoControl 0x000f3438
IQueueCallbackWrite 0x000f3440

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfdownkmirp

Article • 04/03/2024

The **!wudfext.downkmmirp** extension displays the kernel-mode I/O request packet (IRP) that corresponds to the specified user-mode I/O request packet (UM IRP).

dbgcmd

!wudfext.wudfdownkmirp Address

Parameters

Address

Specifies the address of the UM IRP whose corresponding kernel-mode IRP is to be displayed.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

You can use the [!wudfext.umirps](#) extension command to display a list of all outstanding UM IRPs in the host process.

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfdriverinfo

Article • 04/03/2024

The **!wudfext.wudfdriverinfo** extension displays information about a UMDF driver within the current host process.

```
dbgcmd
```

```
!wudfext.wudfdriverinfo Name
```

Parameters

Name

Specifies the name of the UMDF driver to display information about.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The **!wudfext.wudfdriverinfo** extension iterates through each level in each device stack and displays the driver and device information for each entry that matches the driver whose name is specified in the *Name* parameter.

You can use **!wudfext.wudfdriverinfo** to quickly find the device object for your driver.

The following is an example of the **!wudfext.wudfdriverinfo** display:

```
dbgcmd
```

```
kd> !wudfdriverinfo wudfchodriver
IWDFDriver: 0xf2db8
!WDFDEVICE 0xf2f80
!devstack 0x34e4e0 @ level 0
```

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfdumpobjects

Article • 04/03/2024

The **!wudfext.wudfdumpobjects** extension displays outstanding UMDF objects.

```
dbgcmd
```

```
!wudfext.wudfdumpobjects ObjTrackerAddress
```

Parameters

ObjTrackerAddress

Specifies the address to track leaked objects. This address is displayed in the driver-stop message in the debugger when a leak occurs.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

If the UMDF object tracking option (**TrackObjects**) has been enabled in WDF Verifier, you can use **!wudfext.wudfdumpobjects** to see any leaked objects that remain after the driver unloads.

If the **TrackObjects** option has been enabled, the address of the object tracker is automatically displayed when a leak is detected. Use this address as *ObjTrackerAddress* when executing **!wudfext.wudfdumpobjects**.

This extension can be used at any time, even if UMDF has not broken in to the debugger.

If UMDF is version 1.9 or above, you can use either [!wudfext.umdevstack](#) or [!wudfext.umdevstacks](#) to determine the address of the object tracker. This address can then be passed to **!wudfext.wudfdumpobjects**. Here is an example:

dbgcmd

```
0: kd> !umdevstacks
Number of device stacks: 1
Device Stack: 0x038c6f08      Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
    Device 0
        Driver Config Registry Path: WUDFOsrUsbFx2
        UMDriver Image Path:
D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
    Fx Driver: IWDFDriver 0x3076ff0
    Fx Device: IWDFDevice 0x3082e70
        IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0306eff8
    Open UM files (use !umfile <addr> for details):
        0x04a8ef84
    Device XferMode: CopyImmediately RW: Buffered CTL: Buffered
    Object Tracker Address: 0x03074fd8
        Object Tracking ON
        Refcount Tracking OFF
    DevStack XferMode: CopyImmediately RW: Buffered CTL: Buffered
```

```
0: kd> !wudfdumpobjects 0x03074fd8
WdfTypeDriver      Object: 0x03076fb0, Interface: 0x03076ff0
WdfTypeDevice     Object: 0x03082e30, Interface: 0x03082e70
WdfTypeIoTarget   Object: 0x03088f50, Interface: 0x03088f90
WdfTypeIoQueue       Object: 0x0308ce58, Interface: 0x0308ce98
WdfTypeIoQueue       Object: 0x03090e58, Interface: 0x03090e98
WdfTypeIoQueue       Object: 0x03092e58, Interface: 0x03092e98
WdfTypeIoTarget   Object: 0x03098f40, Interface: 0x03098f80
WdfTypeFile        Object: 0x0309cfa0, Interface: 0x0309cfe0
WdfTypeUsbInterface Object: 0x030a0f98, Interface: 0x030a0fd8
WdfTypeRequest    Object: 0x030a2ef8, Interface: 0x030a2f38
WdfTypeIoTarget   Object: 0x030a6f30, Interface: 0x030a6f70
WdfTypeIoTarget   Object: 0x030aaaf30, Interface: 0x030aaaf70
WdfTypeIoTarget   Object: 0x030aef30, Interface: 0x030aef70
WdfTypeRequest    Object: 0x030c6ef8, Interface: 0x030c6f38
WdfTypeRequest    Object: 0x030ceef8, Interface: 0x030cef38
WdfTypeMemoryObject Object: 0x030d6fb0, Interface: 0x030d6ff0
WdfTypeMemoryObject Object: 0x030dcfb0, Interface: 0x030dcff0
WdfTypeFile        Object: 0x030e4fa8, Interface: 0x030e4fe8
WdfTypeFile        Object: 0x030e6fa8, Interface: 0x030e6fe8
WdfTypeFile        Object: 0x030e8fa8, Interface: 0x030e8fe8
WdfTypeRequest    Object: 0x030eaef8, Interface: 0x030eaf38
WdfTypeMemoryObject Object: 0x030ecfb0, Interface: 0x030ecff0
WdfTypeMemoryObject Object: 0x030eefb0, Interface: 0x030eef0
```

Feedback

Was this page helpful?

 Yes

 No

!wudfext.wudffile

Article • 04/03/2024

The **!wudfext.wudffile** extension displays information about a framework file.

dbgcmd

```
!wudfext.wudffile pWdffile [TypeName]
```

Parameters

pWdffile

Specifies the address of the **IWDFFile** interface to display information about.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudffilehandletarget

Article • 04/03/2024

The **!wudfext.wudffilehandletarget** extension displays information about a file-handle-based I/O target.

dbgcmd

```
!wudfext.wudffilehandletarget pWDFFileHandleTarget TypeName
```

Parameters

pWDFFileHandleTarget

Specifies the address of the **IWDFIoTarget** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFIoTarget**.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfiotarget

Article • 04/03/2024

The **!wudfext.wudfiotarget** extension displays information about an I/O target including the target's state and list of sent requests.

dbgcmd

```
!wudfext.wudfiotarget pWDFTarget TypeName
```

Parameters

pWDFTarget

Specifies the address of the **IWDFIoTarget** interface to display information about. The **!wudfext.wudfobject** extension command determines the address of **IWDFIoTarget**.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfobject

Article • 10/25/2023

The **!wudfext.wudfobject** extension displays information about a WDF object, as well as its parent and child relationships.

dbgcmd

```
!wudfext.wudfobject pWDFObject Flags TypeName
```

Parameters

pWDFObject

Specifies the address of the WDF interface to display information about.

Flags

Optional. Specifies the type of information to be displayed. *Flags* can be any combination of the following bits. The default value is 0x01.

Bit 0 (0x01)

Steps recursively through the object hierarchy to obtain the parent and child relationships, which are displayed.

Bit 1 (0x02)

Displays only summary information about the object.

Bit 8 (0x80)

Steps recursively through the object hierarchy, and displays details about the internal framework.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Windows 2000	Unavailable
Windows XP and later	Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

You can use **!wudfext.wudfobject** to list, for example, the child objects of an **IWDFDevice** object, which generally include the device's queues.

You can also use **!wudfext.wudfobject** to find WDF objects that are associated with a particular device, to check the state of a WDF object (for example, whether the WDF object is in the process of deletion), or to determine the WDF object's current reference count.

The **!wudfext.wudfobject** extension also displays the callback functions and context objects that the driver associated with each framework object and attempts to determine the framework object's type. This last feature might not work with certain compilers.

The following are some examples. In the first example, **!wudfext.umdevstacks** gives 0x03050E70 as the address of a device object, and this address is then passed to **!wudfext.wudfobject**. The 0x1 flag is included to display all the children of this object.

```
dbgcmd

0: kd> !umdevstacks
Number of device stacks: 1
  Device Stack: 0x038f6f08    Pdo Name: \Device\USBPDO-11
    Number of UM devices: 1
      Device 0
        Driver Config Registry Path: WUDFOsrUsbFx2
        UMDriver Image Path:
          D:\Windows\system32\DRIVERS\UMDF\WUDFOsrUsbFx2.dll
        Fx Driver: IWDFDriver 0x3044ff0
        Fx Device: IWDFDevice 0x3050e70
          IDriverEntry: WUDFOsrUsbFx2!CMyDriver 0x0303eff8
        Open UM files (use !umfile <addr> for details):
          0x049baf84
        Device XferMode: CopyImmediately RW: Buffered CTL: Buffered
        Object Tracker Address: 0x00000000
          Object Tracking OFF
          Refcount Tracking OFF
        DevStack XferMode: CopyImmediately RW: Buffered CTL: Buffered

0: kd> !wudfobject 0x3050e70 1
IWDFDevice 0x3050e70 Fx: 0x3050e30 [Ref 2]
  15 Children
    00: IWDFIoTarget 0x3056f90 Fx: 0x3056f50 [Ref 3]
```

```
        No Children
01: <Internal>
02: <Internal>
03: <Internal>
04: IWDFIoQueue 0x305ae98 Fx: 0x305ae58 [Ref 8]
        No Children
05: IWDFIoQueue 0x305ee98 Fx: 0x305ee58 [Ref 2]
        No Children
06: IWDFIoQueue 0x3060e98 Fx: 0x3060e58 [Ref 2]
        No Children
07: IWDFIoTarget 0x3066f80 Fx: 0x3066f40 [Ref 2]
        1 Children
            00: IWDFUsbInterface 0x306efd8 Fx: 0x306ef98 [Ref 1]
            3 Children
                00: IWDFIoTarget 0x3074f70 Fx: 0x3074f30 [Ref 2]
                2 Children
                    00: IWDFMemory 0x30a4ff0 Fx: 0x30a4fb0 [Ref 2]
                    No Children
                    01: IWDFMemory 0x30aaff0 Fx: 0x30aafb0 [Ref 2]
                    No Children
                01: IWDFIoTarget 0x3078f70 Fx: 0x3078f30 [Ref 1]
                No Children
```

Here is an example of !wudfext.wudfobject displaying a file object:

```
dbgcmd

kd> !wudfobject 0xf5060
IWDFFile 0xf5060 Fx: 0xf4fe8 [Ref 1]
State: Created Parent: 0xf2f80
No Children
```

Here is an example of !wudfext.wudfobject displaying a driver object:

```
dbgcmd

kd> !wudfobject 0xf2db8 0x01
IWDFDriver 0xf2db8 Fx: 0xf2d40 [Ref 2]
Callback: (WUDFEchoDriver!CMyDriver, 0xf2c68)
State: Created Parent: 0
1 Children:
    00: IWDFDevice 0xf2f80 Fx: 0xf2f08 [Ref 2]
        State: Created Parent: 0xf2db8
        5 Children:
            00: IWDFIoTarget 0xf33c0 Fx: 0xf3348 [Ref 3]
                State: Created Parent: 0xf2f80
                No Children
            01: IWDFIoQueue 0xf3500 Fx: 0xf3488 [Ref 3]
                State: Created Parent: 0xf2f80
                No Children
            02: IWDFFile 0xf5060 Fx: 0xf4fe8 [Ref 1]
                State: Created Parent: 0xf2f80
```

```
No Children
03: IWDFFile 0xf5100 Fx: 0xf5088 [Ref 101]
    State: Created    Parent: 0xf2f80
    No Children
04: IWDFFile 0xf51a0 Fx: 0xf5128 [Ref 101]
    State: Created    Parent: 0xf2f80
    No Children
```

!wudfext.wudfqueue

Article • 04/03/2024

The **!wudfext.wudfqueue** extension displays information about an I/O queue.

```
dbgcmd
```

```
!wudfext.wudfqueue pWDFQueue
```

Parameters

pWDFQueue

Specifies the address of the **IWDFIoQueue** interface to display information about. The [!wudfext.wudfdevicequeues](#) extension command determines the address of **IWDFIoQueue**.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The following is an example of the **!wudfext.wudfqueue** display:

```
dbgcmd
```

```
kd> !wudfqueue 0x000f3500
    WdfIoQueueDispatchSequential, POWER MANAGED, WdfIoQueuePowerOn, CAN
    ACCEPT, CAN DISPATCH
        Number of driver owned requests: 1
            IWDFIoRequest 0x000fa7c0      CwdfIoRequest 0x000fa748
        Number of waiting requests: 199
            IWDFIoRequest 0x000fa908      CwdfIoRequest 0x000fa890
            IWDFIoRequest 0x000faa50      CwdfIoRequest 0x000fa9d8
            IWDFIoRequest 0x000fab98      CwdfIoRequest 0x000fab20
            ...
            IWDFIoRequest 0x000fa530      CwdfIoRequest 0x000fa4b8
            IWDFIoRequest 0x000fa678      CwdfIoRequest 0x000fa600
```

Driver's callback interfaces.
IQueueCallbackRead 0x000f343c
IQueueCallbackDeviceIoControl 0x000f3438
IQueueCallbackWrite 0x000f3440

Feedback

Was this page helpful?



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

!wudfext.wudfrefhist

Article • 10/25/2023

The **!wudfext.wudfrefhist** extension displays the reference count stack history for a UMDF object.

dbgcmd

```
!wudfext.wudfrefhist ObjectAddress
```

Parameters

ObjectAddress

Specifies the address of the UMDF object whose reference count stack history is to be displayed. Note that this is the address of the object itself, not of the interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The **!wudfext.wudfrefhist** command is not supported by UMDF 1.11.

The *ObjectAddress* parameter must be the address of the UMDF object, not the address of the interface (which is used by many other UMDF extension commands). To determine the address of the UMDF object, use the [**!wudfext.wudfdumpobjects**](#) command, which displays both the UMDF object address and the interface address. Alternatively, if you know the address of the interface, you can use it as the argument of the [**!wudfext.wudfobject**](#) command, which displays the object address (displayed after the symbol "Fx:").

If the reference count tracking option ([TrackRefCounts](#)) has been enabled in WDF Verifier, you can use **!wudfext.wudfrefhist** to display each call stack that increments or decrements an object's reference count. You can determine whether a call stack is

causing a memory leak by examining its **AddRef** and **Release** calls for references that are being added and not released.

This command can be used at any time, even if UMDF has not broken in to the debugger.

If this command does not display the desired information, make sure that the relevant data is paged in and then try again.

Here is an example of using the **!wudfext.wudfrefhist** command:

```
dbgcmd

0:007> !wudfext.umdevstacks
...
      UMDriver Image Path:
C:\Windows\System32\drivers\UMDF\WUDFOsrUsbFilter.dll
      Object Tracker Address: 0x0000003792ee9fc0
          Object Tracking ON
          Refcount Tracking ON

0:007> !wudfext.wudfdumpobjects 0x0000003792ee9fc0
...
WdfTypeIoQueue    Object: 0x0000003792f05ce0, Interface: 0x0000003792f05d58

## 0:007> !wudfext.wudfrefhist 0x0000003792f05ce0
-----
## 2  (++)

-----
WUDFx!UfxObject::TrackRefCounts+0xb2
WUDFx!CWdfIoQueue::AddRef+0x17adc
WUDFx!CwdfIoQueue::CreateAndInitialize+0xeb
WUDFx!CwdfDevice::CreateIoQueue+0x1e7
WUDFOsrUsbFilter!CMyQueue::Initialize+0x48
WUDFOsrUsbFilter!CMyDevice::Configure+0x7d
WUDFOsrUsbFilter!CMyDriver::OnDeviceAdd+0xca
WUDFx!CwdfDriver::OnAddDevice+0x486
WUDFx!CWUDF::AddDevice+0x43
WUDFHost!CwudfDeviceStack::LoadDrivers+0x320
WUDFHost!CLpcNotification::Message+0x1340
WUDFPlatform!WdfLpcPort::ProcessMessage+0x140
WUDFPlatform!WdfLpcCommPort::ProcessMessage+0x92
WUDFPlatform!WdfLpc::RetrieveMessage+0x20c
```

!wudfext.wudfreuest

Article • 10/25/2023

The **!wudfext.wudfreuest** extension displays information about an I/O request.

```
dbgcmd
```

```
!wudfext.wudfreuest pWDFRequest
```

Parameters

pWDFRequest

Specifies the address of the **WDFIoRequest** interface to display information about. The [!wudfext.wudfqueue](#) extension command determines the address of **WDFIoRequest**.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Remarks

The following is an example of the **!wudfext.wudfreuest** display:

```
dbgcmd
```

```
kd> !wudfreuest 0x000fa530
CwdfIoRequest 0x000fa4b8
Type: WdfRequestRead
  IWDFIoQueue: 0x000f3500
Completed: No
Canceled: No
UM IRP: 0x00429108 UniqueId: 0xf4 Kernel Irp: 0x0x936ef160
  Type: WudfMsg_READ
  ClientProcessId: 0x1248
  Device Stack: 0x003be4e0
  IoStatus
    hrStatus: 0x0
    Information: 0x0
  Driver/Framework created IRP: No
```

```
Data Buffer: 0x00000000 / 0
IsFrom32BitProcess: Yes
CancelFlagSet: No
Cancel callback: 0x000fa534
Total number of stack locations: 2
CurrentStackLocation: 2 (StackLocation[ 1 ])
    StackLocation[ 0 ]
        UNINITIALIZED
    > StackLocation[ 1 ]
        IWDFRequest:  ****
        IWDFDevice:  0x000f2f80
        IWDFFile:   0x00418cf0
        Completion:
            Callback:  0x00000000
            Context:   0x00000000
        Parameters: (RequestType: WdfRequestRead)
            Buffer length:      0x400
            Key:                0x00000000
            Offset:              0x0
```

!wudfext.wudfusbinterface

Article • 04/03/2024

The **!wudfext.wudfusbinterface** extension displays information about a USB interface object.

dbgcmd

```
!wudfext.wudfusbinterface pWDFUSBInterface TypeName
```

Parameters

pWDFUSBInterface

Specifies the address of the **IWDFUsbInterface** interface to display information about.

The **!wudfext.wudfobject** extension command determines the address of **IWDFUsbInterface**.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfusbpipe

Article • 04/03/2024

The **!wudfext.wudfusbpipe** extension displays information about a USB pipe object.

dbgcmd

```
!wudfext.wudfusbpipe pWDFUSBPipe TypeName
```

Parameters

pWDFUSBPipe

Specifies the address of the **IWDFUsbTargetPipe** interface to display information about.

The **!wudfext.wudfobject** extension command determines the address of

IWDFUsbTargetPipe.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wudfext.wudfusbtarget

Article • 04/03/2024

The **!wudfext.wudfusbtarget** extension displays information about a USB I/O target.

dbgcmd

```
!wudfext.wudfusbtarget pWDFUSBTarget TypeName
```

Parameters

pWDFUSBTarget

Specifies the address of the **IWDFUsbTargetDevice** interface to display information about. The [!wudfext.wudfobject](#) extension command determines the address of **IWDFUsbTargetDevice**.

TypeName

Optional. Specifies the type of the interface (for example, **IWDFDevice**). If a value for *TypeName* is supplied, the extension uses the value as the type of the interface. If an asterisk (*) is supplied as *TypeName*, or if *TypeName* is omitted, the extension attempts to automatically determine the type of the supplied interface.

DLL

Wudfext.dll

Additional Information

For more information, see [User-Mode Driver Framework Debugging](#).

Feedback

Was this page helpful?

 Yes

 No

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

WMI Tracing Extensions (Wmitrace.dll)

Article • 10/25/2023

The extension commands for software tracing sessions can be found in Wmitrace.dll, a library of functions designed to use Windows Management Instrumentation (WMI) for event tracing.

!wmitrace.disable

Article • 04/03/2024

The **!wmitrace.disable** extension disables a provider for the specified Event Tracing for Windows (ETW) trace session.

dbgcmd

```
!wmitrace.disable { LoggerID | LoggerName } GUID
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUID

Specifies the GUID of the provider to be disabled.

DLL

Wmitrace.dll

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the [g \(Go\)](#) command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To enable a provider, use [!wmitrace.enable](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.dumpmini

Article • 04/03/2024

The `!wmitrace.dumpmini` extension displays the system trace fragment, which is stored in a dump file.

```
dbgcmd
```

```
!wmitrace.dumpmini
```

DLL

Wmitrace.dll

This extension is available in Windows Vista and later versions of Windows.

This extension is useful only when debugging a minidump file or a full dump file.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

The *system trace fragment* is a copy of the contents of the last buffer of the System Context Log. Under normal conditions, this is the trace session whose logger ID is 2.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.dumpminievent

Article • 04/03/2024

The **!wmitrace.dumpminievent** extension displays the system event log trace fragment, which is stored in a dump file.

dbgcmd

!wmitrace.dumpminievent

DLL

Wmitrace.dll

This extension is available in Windows Vista Service Pack 1 (SP1) and later versions of Windows.

This extension is useful only when debugging a minidump file or a full dump file.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

The *system event log trace fragment* is a copy of the contents of the last buffer of the System Event Log. The **!wmitrace.dumpminievent** extension displays its contents in event log format.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.dynamicprint

Article • 10/25/2023

The **!wmitrace.dynamicprint** extension controls whether the debugger displays the trace messages generated by a session running in KD_FILTER_MODE.

dbgcmd

```
!wmitrace.dynamicprint {0 | 1}
```

Parameters

0

Turns the trace message display off.

1

Turns the trace message display on.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For help in starting a trace session, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

Before you use this extension, start a trace session, and specify that the trace messages should be sent to the debugger. For example, if you use [!wmitrace.start](#) to start the session, use the **-kd** parameter. If you use Tracelog to start the trace session, use its **-kd** parameter. Tracelog (tracelog.exe) is a trace controller included in the Windows Driver Kit.

Trace messages are held in buffers on the target computer. Those buffers are flushed and sent to the debugger on the host computer at regular intervals. You can specify the flush timer interval by using the **-kd** parameter of the [!wmitrace.start](#) command or the **-kd** parameter of the Tracelog tool. Starting in Windows 8, you can specify the flush timer value in milliseconds by appending **ms** to the flush timer value.

By default, ETW maintains per-processor trace buffers on the target computer. When the trace buffers are flushed and sent to the debugger on the host computer, there is no mechanism for merging the buffers into a chronological sequence of events. So the events might be displayed out of order. Starting in Windows 7, you can solve this problem by setting the **-lowcapacity** parameter when you use the Tracelog tool to start a trace session.

Tracelog MySession -kd -lowcapacity

When you start a session with **-lowcapacity** set, all events go to a single buffer on the target computer, and the events are displayed in the correct order in the debugger on the host computer.

Also, before using this extension, use [!wmitrace.searchpath](#) or [!wmitrace.tmffile](#) to specify the trace message format files. The system uses the trace message format files to format the binary trace messages so that they can be displayed as human-readable text.

See also

[!wmitrace.start](#)

!wmitrace.enable

Article • 04/03/2024

The **!wmitrace.enable** extension enables a provider for the specified Event Tracing for Windows (ETW) trace session.

dbgcmd

```
!wmitrace.enable { LoggerID | LoggerName } GUID [-level Num] [-matchallkw  
Num] [-matchanykw Num] [-enableproperty Num] [-flag Num]
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUID

Specifies the GUID of the provider to be enabled.

-level *Num*

Specifies the level. *Num* can be any integer.

-matchallkw *Num*

Specifies one or more keywords. If multiple keywords are specified, the provider will be enabled only if all keywords are matched. *Num* can be any integer.

-matchanykw *Num*

Specifies one or more keywords. If multiple keywords are specified, the provider will be enabled if at least one keyword is matched. *Num* can be any integer. The effects of this parameter overlap with the effects of the **-flag** parameter.

-enableproperty *Num*

Specifies the enable property. *Num* can be any integer.

-flag *Num*

Specifies one or more flags. *Num* can be any integer. The effects of this parameter overlap with the effects of the **-matchanykw** parameter.

DLL

Wmitrace.dll

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the **g (Go)** command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To disable a provider, use [!wmitrace.disable](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.eventlogdump

Article • 04/03/2024

The **!wmitrace.eventlogdump** extension displays the contents of the specified logger. The display is formatted like an event log.

dbgcmd

```
!wmitrace.eventlogdump { LoggerID | LoggerName }
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

Wmitrace.dll

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

This extension is similar to the [!wmitrace.logdump](#) extension, except that the output of **!wmitrace.eventlogdump** is formatted in event log style, and the output of **!wmitrace.logdump** is formatted in Windows software trace preprocessor (WPP) style.

You should choose the extension whose format is appropriate for the data you wish to display.

To find the logger ID of a trace session, use the [!wmitrace.strdump](#) extension.
Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.help

Article • 04/03/2024

The `!wmitrace.help` extension displays the extension commands in Wmitrace.dll.

```
dbgcmd
!wmitrace.help
```

DLL

Wmitrace.dll

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.logdump

Article • 10/25/2023

The **!wmitrace.logdump** extension displays the contents of the trace buffers for a trace session. You can limit the display to trace messages from specified providers.

dbgcmd

```
!wmitrace.logdump [-t Count] [{LoggerID|LoggerName} [GUIDFile]]
```

Parameters

-t *Count*

Limits the output to the most recent messages. *Count* specifies the number of messages to display.

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, the trace session with ID equal to 1 is used.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

GUIDFile

Displays only trace messages from providers specified in the *GUIDFile* file. *GUIDFile* represents the path (optional) and file name of a text file that contains the control GUIDs of one or more trace providers, such as a .guid or .ctl file.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

During Windows software trace preprocessor (WPP) software tracing, trace session buffers are used to store trace messages until they are flushed to a log file or to a trace consumer for a real-time display. The **!wmitrace.logdump** extension displays the contents of the buffers that are in physical memory. The display appears in the Debugger Command window.

This extension is especially useful to recover the most recent traces when a crash occurs, and to display the traces stored in a crash dump file.

Before you use this extension, use **!wmitrace.searchpath** or **!wmitrace.tmffile** to specify the trace message format files. The system uses the trace message format files to format the binary trace messages in the buffers so that they can be displayed as human-readable text.

Note If your driver uses UMDF version 1.11 or later, you do not need to use **!wmitrace.searchpath** or **!wmitrace.tmffile**.

When you use Tracelog to start a trace session with circular buffering (-buffering), use this extension to display the buffer contents.

To find the logger ID of a trace session, use the **!wmitrace.strdump** extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension does not display their contents.

This extension is similar to the **!wmitrace.eventlogdump** extension, except that the output of **!wmitrace.logdump** is formatted in WPP style, and the output of **!wmitrace.eventlogdump** is formatted in event log style. You should choose the extension whose format is appropriate for the data you want to display.

For information about how to view the UMDF trace log, see [Using WPP Software Tracing in UMDF-based Drivers](#).

!wmitrace.logger

Article • 04/03/2024

The **!wmitrace.logger** extension displays data about the trace session, including the session configuration data. This extension does not display trace messages generated during the session.

dbgcmd

```
!wmitrace.logger [ LoggerID | LoggerName ]
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, the trace session with ID equal to 1 is used.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

Wmitrace.dll

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK.

Remarks

This extension is designed for performance logs and events, which cannot be formatted for human-readable display. To display the trace messages in a trace session buffer,

along with header data, use [!wmitrace.logdump](#).

To find the logger ID of a trace session, use the [!wmitrace.strdump](#) extension. Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

Feedback

Was this page helpful?



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

!wmitrace.logsav

Article • 04/03/2024

The **!wmitrace.logsav** extension writes the current contents of the trace buffers for a trace session to a file.

dbgcmd

```
!wmitrace.logsav {LoggerID|LoggerName} Filename
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

*Filenam*e

Specifies a path (optional) and file name for the output file.

DLL

Wmitrace.dll

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see "Tracelog" in the Windows Driver Kit (WDK).

Remarks

This extension displays only the traces that are in memory at the time. It does not display trace messages that have been flushed from the buffers and delivered to an event trace log file or to a trace consumer.

Trace session buffers store trace messages until they are flushed to a log file or to a trace consumer for a real-time display. This extension saves the contents of the buffers that are in physical memory to the specified file.

The output is written in binary format. Typically, these files use the .etl (event trace log) filename extension.

When you use Tracelog to start a trace session with circular buffering (-buffering), you can use this extension to save the current buffer contents.

To find the logger ID of a trace session, use the [!wmitrace.strdump](#) extension.

Alternatively, you can use the Tracelog command tracelog -l to list the trace sessions and their basic properties, including the logger ID.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.searchpath

Article • 10/25/2023

The **!wmitrace.searchpath** extension specifies the location of the trace message format files for messages in the trace buffers.

dbgcmd

```
!wmitrace.searchpath [+ TMFPath  
!wmitrace.searchpath
```

Parameters

+

Causes *TMFPath* to be appended to the existing search path. If the plus (+) token is not used, *TMFPath* replaces the existing search path.

TMFPath

The path to the directory where the debugger should look for the trace message format files. Paths that contain spaces are not supported. If multiple paths are included, they should be separated by semicolons, and the entire string should be enclosed in quotation marks. If the path is in quotation marks, the backslash character must be preceded by an escape character ("c:\\debuggers;c:\\debuggers2"). When the + token is used, *TMFPath* will be appended to the existing path, with a semicolon automatically inserted between the existing path and the new path; however, if the + token is used, quotation marks cannot be used.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about trace message format files, see the "Trace Message Format Files"

topic in the Windows Driver Kit (WDK).

Remarks

When used with no parameters, **!wmitrace.searchpath** displays the current search path.

The trace message format files (*.tmf) contain instructions for formatting the binary trace messages that a trace provider generates.

The *TMFPath* parameter must contain only a path to a directory; it cannot include a file name. The name of a TMF file is a message GUID followed by the .tmf extension. When the system formats a message, it reads the message GUID on the message and searches recursively for a TMF file whose name matches the message GUID, beginning in the specified directory.

Windows needs a TMF file in order to format the binary trace messages in a buffer. Use **!wmitrace.searchpath** or **!wmitrace.tmffile** to specify the TMF file before using **!wmitrace.dynamicprint** or **!wmitrace.logdump** to display trace buffer contents.

If you do not use either **!wmitrace.searchpath** or **!wmitrace.tmffile**, the system uses the value of the TRACE_FORMAT_SEARCH_PATH environment variable. If that variable is not present, it uses the default.tmf file, which is included in Windows. If the system cannot find any formatting information for a trace message, it writes a "No format information found" error message in place of the trace message content.

!wmitrace.setprefix

Article • 10/25/2023

The `!wmitrace.setprefix` extension specifies the trace message prefix that is prepended to the trace messages from this session. This extension allows you to change the prefix during the debugging session.

dbgcmd

```
!wmitrace.setprefix [+]
!wmitrace.setprefix
```

Parameters

+

Causes *PrefixVariables* to be appended to the trace message prefix. If the + token is not used, *PrefixVariables* replaces the existing trace message prefix.

PrefixVariables

A set of variables that specifies the format and data in the trace message prefix.

The variables have the format %n!x!, where %n represents a data field and !x! represents the data type. You can also include separation characters, such as colons (:), semicolons (;), parentheses (()), braces ({ }), and brackets ([]) to separate the fields.

Each %n variable represents a parameter that is described in the following table.

Prefix variable identifier	Variable type	Description
%1	string	The friendly name of the message GUID of the trace message. By default, the friendly name of a message GUID is the name of the directory in which the trace provider was built.
%2	string	Source file and line number. This variable represents the friendly name of the trace message. By default, the friendly name of a trace message is the name of the

Prefix variable identifier	Variable type	Description
		source file and the line number of the code that generated the trace message.
%3	ULONG	Thread ID. Identifies the thread that generated the trace message.
%4	string	Time stamp of the time that the trace message was generated.
%5	string	Kernel time. Displays the elapsed execution time for kernel-mode instruction, in CPU ticks, at the time that the trace message was generated.
%6	string	User time. Displays the elapsed execution time for user-mode instruction, in CPU ticks, at the time that the trace message was generated.
%7	LONG	Sequence number. Displays the local or global sequence number of the trace message. Local sequence numbers, which are unique only to this trace session, are the default.
%8	ULONG	Process ID. Identifies the process that generated the trace message.
%9	ULONG	CPU number. Identifies the CPU on which the trace message was generated.
%!FUNC!	string	Function name.

Prefix variable identifier	Variable type	Description
		Displays the name of the function that generated the trace message.
%!FLAGS!	string	Displays the name of the trace flags that enable the trace message.
%!LEVEL!	string	Displays the value of the trace level that enables the trace message.
%!COMPNAME!	string	Component name. Displays the name of the component of the provider that generated the trace message. The component name appears only if it is specified in the tracing code.
%!SUBCOMP!	string	Subcomponent name. Displays the name of the subcomponent of the provider that generated the trace message. The subcomponent name appears only if it is specified in the tracing code.

The symbol within exclamation marks is a conversion character that specifies the format and precision of the variable. For example, %8!04X! specifies the process ID formatted as a four-digit, unsigned, hexadecimal number.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Example

The following command changes the trace message prefix in the debugger output to the following format:

```
!wmitrace.setprefix %2!s!: %!FUNC!: %8!04x!.%3!04x!: %4!s!:
```

This extension command sets the trace message prefix to the following format:

SourceFileLineNumber: FunctionName: ProcessID.ThreadID: SystemTime:

As a result, the trace messages are prepended with the specified information in the specified format. The following code example is taken from a trace of the Tracedrv sample driver in the WDK.

```
dbgcmd
```

```
tracedrv_c258: TracedrvDispatchDeviceControl: 0af4.0c64: 07/25/2003-  
13:55:39.998: IOCTL = 1
```

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK documentation. For information about trace message format files, see the "Trace Message Prefix" topic in the WDK documentation.

Remarks

When used with no parameters, **!wmitrace.setprefix** displays the current value of the trace message prefix.

The *trace message prefix* consists of data about the trace message that is prepended to each trace message during Windows software trace preprocessor (WPP) software tracing. This data originates in the trace log (.etl) file and the trace message format (.tmf) file. You can customize the format and data in trace message prefix.

The default trace message prefix is as follows:

```
dbgcmd
```

```
[%9!d!]%8!04X!.%3!04X!::%4!s! [%1!s!]
```

and produces the following prefix:

```
dbgcmd
```

```
[CPUNumber]ProcessID.ThreadID::SystemTime [ProviderDirectory]
```

You can change the format and data in the trace message prefix outside of the debugger by setting the %TRACE_FORMAT_PREFIX% environment variable. For an example that illustrates how to set the trace message prefix outside of the debugger, see "Example 7: Customizing the Trace Message Prefix" in the Windows Driver Kit (WDK) documentation. If the trace message prefix of your messages varies from the default, this environment variable might be set on your computer.

The prefix that you set by using this extension command affects only the debugger output. The trace message prefix that appears in the trace log is determined by the default value and the value of the %TRACE_FORMAT_PREFIX% environment variable.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension does not affect them.

!wmitrace.start

Article • 10/25/2023

The **!wmitrace.start** extension starts the Event Tracing for Windows (ETW) logger on the target computer.

dbgcmd

```
!wmitrace.start LoggerName [-cir Size | -seq Size] [-f File] [-b Size] [-max Num] [-min Num] [-kd] [-ft Time]
```

Parameters

LoggerName

Supplies a name to be used for the trace session. *LoggerName* cannot contain spaces or quotation marks.

-cir *Size*

Causes the log file to be written in a circular manner. *Size* specifies the maximum file size, in bytes. When the file reaches this length, new data will be written to the file in a circular manner, overwriting the file from beginning to end. This cannot be combined with the **-seq** parameter. If neither **-cir** nor **-seq** is specified, the file is written in buffered mode.

-seq *Num*

Causes the log file to be written in a sequential manner. *Size* specifies the maximum file size, in bytes. When the file reaches this length, the oldest data will be deleted from the beginning of the file whenever new data is appended to the end. This cannot be combined with the **-cir** parameter. If neither **-cir** nor **-seq** is specified, the file is written in buffered mode.

-f *File*

Specifies the name of the log file to be created on the target computer. *File* must include an absolute directory path, and cannot contain spaces or quotation marks.

-b *Size*

Specifies the size of each buffer, in kilobytes. The permissible range of *Size* is between 1 and 2048, inclusive.

-max *Num*

Specifies the maximum number of buffers to use. *Num* can be any positive integer.

-min *Num*

Specifies the minimum number of buffers to use. *Num* can be any positive integer.

-kd

Enables KD filter mode. Messages will be sent to the kernel debugger and displayed on the screen.

-ft *Time*

Specifies the duration of the flush timer, in seconds. Starting in Windows 8, you can specify the flush timer duration in milliseconds by appending **ms** to the *Time* value. For example, **-ft 100ms**.

Note If you start a tracing session in KD filter mode (**-kd**), trace buffers on the target computer are sent to the debugger on the host computer for display. This parameter specifies how often the buffers on the target computer are flushed and sent to the host computer.

DLL

This extension is exported by Wmitrace.dll.

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For more details on the parameters of this extension, see [StartTraceA Function](#) and [EVENT_TRACE_PROPERTIES](#). For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the [g \(Go\)](#) command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

When the trace session is started, the system assigns it an ordinal number (the *logger ID*). The session can then be referred to either by the logger name or the logger ID.

To stop the ETW logger, use [!wmitrace.stop](#).

!wmitrace.stop

Article • 04/03/2024

The **!wmitrace.stop** extension stops the Event Tracing for Windows (ETW) logger on the target computer.

dbgcmd

```
!wmitrace.stop { LoggerID | LoggerName }
```

Parameters

LoggerID

Specifies the trace session. *LoggerID* is an ordinal number that the system assigns to each trace session on the computer.

LoggerName

Specifies the trace session. *LoggerName* is the text name that was specified when the trace session was started.

DLL

Wmitrace.dll

This extension is available in Windows 7 and later versions of Windows.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about tracing tools, see the Windows Driver Kit (WDK).

Remarks

After using this extension, you must resume program execution (for example, by using the [g \(Go\)](#) command) in order for it to take effect. After a brief time, the target computer automatically breaks into the debugger again.

To start the ETW logger, use [!wmitrace.start](#).

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.strdump

Article • 04/03/2024

The **!wmitrace.strdump** extension displays the WMI event trace structures. You can limit the display to the structures for a particular trace session.

dbgcmd

```
!wmitrace.strdump [ LoggerID | LoggerName ]
```

Parameters

LoggerID

Limits the display to the event trace structures for the specified trace session. *LoggerID* specifies the trace session. It is an ordinal number that the system assigns to each trace session on the computer. If no parameter is specified, all trace sessions are displayed.

LoggerName

Limits the display to the event trace structures for the specified trace session.

LoggerName is the text name that was specified when the trace session was started. If no parameter is specified, all trace sessions are displayed.

DLL

Wmitrace.dll

This extension is available in Windows 2000 and later versions of Windows. If you want to use this extension with Windows 2000, you must first copy the Wmitrace.dll file from the winxp subdirectory of the Debugging Tools for Windows installation directory to the w2kfre subdirectory.

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about Tracelog, see the "Tracelog" topic in the Windows Driver Kit (WDK).

Remarks

To find the logger ID of a trace session, use the `!wmitrace.strdump` extension. Alternatively, you can use the Tracelog command `tracelog -l` to list the trace sessions and their basic properties, including the logger ID.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.tmffile

Article • 04/03/2024

The **!wmitrace.tmffile** extension specifies a trace message format (TMF) file. The file specified by this extension is used to format trace messages displayed or written by other WMI tracing extensions.

dbgcmd

```
!wmitrace.tmffile TMFFile
```

Parameters

TMFFile

Specifies a trace message format file.

DLL

Wmitrace.dll

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK. For information about trace message format files, see the "Trace Message Format File" topic in the Windows Driver Kit (WDK).

Remarks

Trace message format files (.tmf) are structured text files that are created during Windows software trace preprocessor (WPP) software tracing. These files contain instructions for formatting trace binary trace messages so that they can be displayed in human-readable form.

In order to display the trace message in a trace buffer ([!wmitrace.logdump](#)) or write them to a file ([!wmitrace.logsav](#)e), you must first identify the TMF files for the trace messages.

You can use [!wmitrace.searchpath](#) to specify a directory in which TMF files are stored. The system then searches the directory for a TMF file that contains instructions for the

messages that it is formatting. (It uses the message GUID to associate the message with the correct TMF file.)

However, you can use `!wmitrace.tmffile` to specify a particular TMF file. You must use `!wmitrace.tmffile` if the TMF file name is not a message GUID followed by the .tmf extension. Otherwise, the system will not find it.

If you do not use either `!wmitrace.searchpath` or `!wmitrace.tmffile`, the system uses the value of the TRACE_FORMAT_SEARCH_PATH environment variable. If that variable is not present, it uses the default.tmf file. If the system cannot find formatting information for a trace message, it writes a "No format information found" error message, instead of the trace message content.

Note If your driver uses UMDF version 1.11 or later, you do not need to use `!wmitrace.searchpath` or `!wmitrace.tmffile`.

This extension is only useful during WPP software tracing, and earlier (legacy) methods of Event Tracing for Windows. Trace events that are produced by other manifested providers do not use trace message format (TMF) files, and therefore this extension cannot be used with them.

Feedback

Was this page helpful?

 Yes

 No

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

!wmitrace.traceoperation

Article • 04/03/2024

The **!wmitrace.traceoperation** extension displays the progress messages from the tracing components in Windows.

dbgcmd

```
!wmitrace.traceoperation {0 | 1 | 2}
```

Parameters

0

Disables the display of tracing progress messages.

1

Enables the display of tracing progress messages. All messages generated by the WMI tracing debugger extensions are displayed.

2

Enables the display of tracing progress messages. All messages generated by the WMI tracing debugger extensions or by Tracefmt are displayed.

DLL

Wmitrace.dll

Additional Information

For a conceptual overview of event tracing, see the Microsoft Windows SDK.

Remarks

This extension causes the tracing components to display verbose output. This feature is useful to troubleshoot software tracing.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A