WEBVIEW2

AUTOIT UDF - DEVELOPER GUIDE

WebView2 developer guide

Chromium inside an AutoIt window, with no file to ship: the COM model, the ABI traps and the diagnostics, explained from the source.

AutoIt 3.3.16+ | No loader to ship | 113 functions | 539 checks | 39 handlers

```
#include "WebView2.au3"

Local $hView = _WV2_Create($hGui)
_WV2_Navigate($hView, "https://example.com")
_WV2_WaitNavigation($hView)

Local $sTitle = _WV2_ExecuteScript($hView, "document.title")
```

One .au3 file. No DLL, no OCX, no installer.

THE WHOLE DEPENDENCY LIST

VERSION **0.6.5**

RUNTIME **AutoIt 3.3.16+**

LICENCE **MIT**

THE STACK

AutoIt script
GUI or --auto suite

▼

WebView2.au3
113 functions, pure AutoIt

▼

EmbeddedBrowserWebView.dll
part of Windows

▼

Edge WebView2 runtime
already installed

BY THE NUMBERS

public functions	113
self-test checks	539 over 7
handler slots	39
concurrent views	16
files to ship	0

The runtime is already on the machine. This package ships no browser, no DLL, no installer: one include, and the engine Windows carries.

COLOPHON

Typeset with PDFX 1.4.0, the author's other UDF.

PDFX - forum topic 213930

Numeric1 - forum profile 118605

Version 0.6.5 - November 2026

HOW TO USE THIS DOCUMENT

Contents & bearings

Eight chapters. Read 1 to 4 to understand the machinery, 5 to 8 to change it without breaking what already works.

01 The COM model in AutoIt 12

02 Handler objects in a global heap 14

03 Vtable ranks and WebView2.h 16

04 x86 and x64: put_Bounds 19

05 Adding a method in six steps 22

06 Diagnosing a failure 24

07 Threading and the message pump 26

08 Limits, pitfalls, references 29

Every entry is clickable; every heading is also a bookmark.

THE PACKAGE IN BRIEF

One AutoIt file embeds the Edge engine that Windows already carries: 75 functions, no DLL, no OCX, no installer. This guide is its maintenance manual.

85
PUBLIC FUNCTIONS
pure AutoIt, no loader

390
SELF-TEST CHECKS
no network, no dialog

35
HANDLER SLOTS
one fake vtable each

0
FILES TO SHIP
nothing to register

READING PATHS

Understand the machinery	01 - 02 - 03 - 04
Extend the wrapper	05 - 06
Ship and maintain	07 - 08

What this document is not

Neither an API reference nor a beginner's tutorial. It is the maintenance manual of the UDF: the machinery, the traps, and the checks that keep both honest.

THREE RULES **before touching the code**

1. **Measure, do not assume**
   The header says by value; the 64-bit wire says pointer. Test the shape, then read the effect.
2. **The header is the fact**
   Slot numbers come from WebView2.h, never from memory or a summary page.
3. **hr = 0 is not proof**
   S_OK says the call arrived. Find an observable consequence before believing it.

0.6.5

THE EIGHT LOTS THIS GUIDE DESCRIBES

What 0.4 to 0.6.5 add

The 0.3 edition of this guide described a wrapper over the base interface and eight event kinds. The two lots that followed took the UDF to the versioned interfaces: your script now answers the page instead of only watching it, and the objects the engine returns are kept, steered and released by the same global heap this guide explains. Lot 0.6 added no interface at all: it took the work out of the close path and put a guard on it. Lot 0.6.1 went back to the interfaces, and to the one part of the view nobody had opened: what it plays, what it shows full-screen, and what it is allowed to reach. Lot 0.6.2 stayed on the create path: asking the engine for a browser and waiting for its answer are now two calls, and one environment is shared by every view that was given the same user-data folder. Lot 0.6.3 is the life of the document: the three moments a navigation goes through, and the close a page asks for, are four more kinds in the queue. Lot 0.6.4 is what a view looks like: the scheme the page believes the system is in, the colour behind a page that has not painted, the dark title bar of the window that hosts it, and the folder the profile downloads to - four doors, and no two of them reached the same way. Lot 0.6.5 is what the browser is told before it exists: nine properties of the environment and the user agent of one view, and the first object the UDF hands the runtime to be read rather than called back.

From 0.3 to 0.6.5, measured

	0.3.0	0.5.0	0.6.1	0.6.2	0.6.3	0.6.4	0.6.5
Public functions	40	75	83 - 7 on the audio surface	85 - 2 on the create path	85 - none added	92 - 7 on the appearances	113 - 21 on the environment
Handler slots per view	14	31	35 - audio, mute, full screen, permission	35 - unchanged	39 - four more	39 - unchanged	39 - unchanged
Pointer cells in the heap	1 120	2 480	2 800	2 800 - unchanged	3 120	3 120 - unchanged	3 120 - the options object sits outside it
Event kinds	8	17	21 - four added in 0.6.1	21 - unchanged	25 - four added	25 - unchanged	25 - unchanged
Error codes	12	13	13 - unchanged	13 - none added	13 - none added	13 - none added	13 - none added
Self-test checks	111	280 over 35 sections	297 + 40 audio + 23 media	390 over 4 suites	437 over 5 suites	488 over 6 suites	539 over 7 suites
Audited slot and IID constants	60	227, against WebView 2.h	249 - zero problems	249 - unchanged	265 - zero problems	273 - zero problems	300 - zero problems

	0.3.0	0.5.0	0.6.1	0.6.2	0.6.3	0.6.4	0.6.5
Concurrent views	16	16	16 - unchanged	16 - unchanged	16 - unchanged	16 - unchanged	16 - unchanged

Lot 0.6.0 in between is the close path: _WV2_DestroyAll, the adaptive pumps in the teardown and the ceilings the self test measures rather than assumes. It added one public function and no interface.

Lot 0.4 - the page asks, your script answers

- **Resource handler** _WV2_SetResourceHandler names one AutoIt function that sees every request the page makes - uri, method, headers - and either lets it through or answers it from memory with a status, headers and a body. No web server, no temporary file, no network.
- **Virtual host** _WV2_SetVirtualHostNameToFolderMapping serves a folder under an invented host such as https://app.local/, which gives the page a real web origin: storage, workers, fetch and CORS behave as they do online.
- **Filters** _WV2_AddResourceFilter and _WV2_RemoveResourceFilter take a uri pattern and a $WV2_RES_* kind, so a handler is entered only for what it cares about.
- **Re-entrancy** A call that would pump messages from inside a handler is refused with $WV2_ERR_BUSY. This is the reason the teardown figure in chapter 7 begins with the resource handler.

Mechanically, lot 0.4 is one handler family and two raw ranks the base tag never reached. The subscription is ICoreWebView2::add_WebResourceRequested at slot 55, the filters are calls 57 and 58, and the body your function returns travels through an IStream whose Read and Seek are slots 3 and 5 of the Windows SDK - not of WebView2.h. Chapter 3 lists them, chapter 5 walks one through end to end.

Lot 0.5 - the versioned interfaces, one by one

- **Find in page** _WV2_FindStart drives the search object reached through _28::get_Find, with case and whole-word options and highlights; _WV2_FindNext and _WV2_FindPrevious step, _WV2_FindCount and _WV2_FindActive read, _WV2_FindStop closes.
- **Cookies** _WV2_GetCookies, _WV2_SetCookie and _WV2_DeleteCookie work on the cookie manager and its cookie objects - name, value, domain, path, expiry, SameSite, HttpOnly, Secure.
- **Printing** _WV2_PrintSettingsCreate builds an ICoreWebView2PrintSettings and _WV2_PrintSettingsSet fills it; _WV2_PrintToPdfEx prints with it, _WV2_PrintToPdfStream returns the PDF in memory, _WV2_ShowPrintUI opens the preview.
- **Downloads** _WV2_GetDownloadCount and _WV2_GetDownload read uri, path, state, bytes and interrupt reason of the eight downloads each view remembers; _WV2_DownloadPause, _WV2_DownloadResume and _WV2_DownloadCancel steer them.
- **Dialogs, keys, menus** _WV2_SetDialogHandler answers alert, confirm, prompt and beforeunload; _WV2_SetAcceleratorHandler sees Ctrl+P before the browser does; _WV2_SetContextMenuHandler takes the menu of the page. All three are handler objects in the same heap.
- **Favicon and profile** _WV2_GetFaviconUri and _WV2_GetFavicon read what _15 declares and bring the picture as PNG or JPEG bytes; _WV2_ClearBrowsingData goes through the _13 profile and takes cookies, cache, history and download records.
- **DevTools protocol** _WV2_CallDevToolsProtocolMethod calls any CDP method and returns its JSON; _WV2_SetDevToolsProtocolEvent subscribes to one CDP event, delivered as $WV2_EVENT_DEVTOOLS.
- **Typed scripts** _WV2_ExecuteScriptEx reads ICoreWebView2ExecuteScriptResult and its exception object, so a script that throws reports the text instead of a null to decode.

Lot 0.6 - the close path

Lot 0.6 adds no face of the engine. It takes work out of the one path a user feels, and puts a guard on it. The measurements are in chapter 7.

- **No delete on the way out** _WV2_Destroy used to DirRemove the temporary user-data folder it owned: 15 542 ms, measured, inside the call, with the window still on the screen. __WV2_Free now queues it and __WV2_SweepStale collects it at the end of the next _WV2_Create, together with the folders of runs that were killed. A folder the caller named is never touched.
- **Adaptive pumps** The fixed pairs of 100 ms pumps in the teardown became __WV2_PumpDrain($iCap, $iQuiet): it returns after $__WV2_DRAIN_QUIET = 20 ms of silence and stops at $__WV2_DRAIN_CAP = 100 ms whatever happens.
- **_WV2_DestroyAll, and no exit handler** The net a script with several views wants is a call in the message loop, not a function registered with OnAutoItExitRegister: a handler runs after the loop the runtime answers through, and the same teardown measured 113 ms in the loop costs 1.4 to 2.7 s there.
- **The guardrails** $WV2_TEST_DESTROY_CAP puts a ceiling on every close in the self test, and section 34 checks the property behind it - the count of this process's profile folders does not change across a close - because a duration can be waved by a busy disk and a property cannot. The ceilings know they are durations: the first create of the run is timed, and past $WV2_TEST_SLOW_MACHINE (12 s, where an idle machine needs 3 to 7) each one reports instead of judging, because a close measured 22 446 ms on this machine with 84% of its memory taken. Sections 36 and 37 cover a profile the application owns and two views released at once.

Lot 0.6.1 - what the page plays, shows and may reach

This lot went back to the header with one question: what does the view publish that no wrapper function had ever read? Three answers, and each one is a different shape of call.

- **The sound is a property of the document** ICoreWebView2_8, slots 81 to 87, a runtime of 97.0.1072.54 or newer. get_IsDocumentPlayingAudio and get_IsMuted are get_ calls on an interface the tag never reaches, so they travel by the raw vtable after a QueryInterface, and their two events are the handler slots 32 and 33. What the property means is the part a wrapper cannot invent: the runtime answers from the samples its pipeline really produces, so digital silence never raises it and --mute-audio holds it at 0 - both measured, and both the reason the self test writes a real 440 Hz WAV instead of trusting an audio tag.
- **The full screen belongs to the base interface** Slots 52 to 54, past get_DocumentTitle where the wrapper tag stops. A get_ on the view pointer itself, no QueryInterface, and so no minimum runtime to name: the two habits this guide keeps recommending - count from the header, and let the HRESULT answer for the version.
- **A permission request the runtime does not answer** add_PermissionRequested is slot 23, inside the tag and the only one of the three that is. put_State settles a request from inside the handler, which is what ALLOW, DENY and DEFAULT do; ASK answers after the handler has returned, and that is what GetDeferral and ICoreWebView2Deferral::Complete exist for. The UDF keeps what it can no longer borrow: __WV2_PermHold takes the deferral, AddRefs the arguments and records the uri and the kind it had already read, because once the callback is over none of those pointers is the host's to trust. _WV2_PermissionRespond completes the deferral later, and the eight slots of $WV2_PERM_MAX are the same bargain the download table strikes: a ninth takes the oldest, unanswered.
- **Two surfaces that already shipped, corrected** _WV2_WaitEvent with $iEvent = 0 pops the oldest event of any type and reports which one in @extended. _WV2_EnableEvent with $WV2_EVENT_DEVTOOLS no longer has to be written after the subscription: it records the intent in $__WV2_bCdpOff, because the subscription carries the event anyway and an explicit disable must still be the answer that sticks.

Lot 0.6.2 - asking for a view is not waiting for one

The measurements that opened chapter 4 said the create takes between one and two seconds and sometimes fifty. A single call cannot serve both, and a fifteen second ceiling sat squarely between them. This lot splits the call rather than guessing a number.

- **The create is two functions** _WV2_CreateStart sends the request and hands back the handle as soon as the engine accepts it - 1 299 ms on the author's machine, where the whole create took 19 527 ms. _WV2_CreateWait then pumps until the callback answers, up to $WV2_CREATE_TIMEOUT. _WV2_Create is exactly those two, so nothing written before changes.
- **A view that is not ready is refused, not guessed** every call on a handle still waiting for its callback answers $WV2_ERR_NOT_READY, and _WV2_Destroy releases it normally. A script can therefore build its window, run its own loop and take the view up when it answers.
- **One environment per user-data folder** __WV2_EnvShareFind looks the folder up before asking for a browser, borrows the environment pointer with an explicit AddRef, and __WV2_EnvShareRetire drops the entry at the last release. The second view on a live folder took 146 ms against the 19 527 ms of the first.
- **The ceiling is now above the worst case** 60 000 ms against 1,2 s for a fresh folder, 25,3 s and 47,6 s for a folder a previous run had already used, and 30 s for a navigation. _WV2_CreateStart takes no timeout at all: it does not wait.

Lot 0.6.3 - the life of a document

Four event kinds and no new public function: ContentLoading, SourceChanged, DOMContentLoaded and WindowCloseRequested ride the handler objects of chapter 2 and the queue of chapter 3. What the lot is really about is where the header puts them, and the measurements are in the fifth suite: 47 checks, 0 failures, on both interpreters.

- **Three ways in, because the interface has three shapes** add_ContentLoading (slot 9) and add_SourceChanged (slot 11) are inside the base wrapper tag. add_WindowCloseRequested is 59, past the end of the tag, so it goes through __WV2_AddRaw. add_DOMContentLoaded is 64 on ICoreWebView2_2, so it goes through the cached QueryInterface of that version and the same raw call. Extending the tag instead would have shifted every later name.
- **ICoreWebView2_2 is not the base interface plus one method** it repeats the base layout, adds its own three calls at 61, 62 and 63, and only then reaches 64. A hand-counted 63 subscribes add_ZoomControlChanged. The audit printed 64 from the official header, and the UDF only uses what the audit printed.
- **The uri comes from the sender, not the args** the lifecycle event objects carry no address, so __WV2_SenderSource calls get_Source - slot 4 of the base layout, which every version repeats - copies the LPWSTR and frees it with CoTaskMemFree.
- **Two BOOLs and one UINT64** IsErrorPage and IsNewDocument are slot 3 of their own arg interfaces, NavigationId is slot 4 of ContentLoading and 3 of DOMContentLoaded. A BOOL is declared int and a UINT64 int64; the id is clamped to 0 rather than answered negative, because $iValue is a signed AutoIt integer.
- **window.close() does nothing** the handler pushes the event and stops: _WV2_IsReady still answers 1 and the title still reads. The limit is Chromium's, not the UDF's - a close is forwarded only when the document is the sole entry of its history, which is why the suite keeps a second view navigated once and asserts the refused case as a timeout.

Lot 0.6.4 - what a view looks like

Seven functions, four doors, and no two of them reached the same way: that is the reason this lot was measured instead of written once and trusted. The numbers are in the sixth suite - 52 checks, 0 failures, on both interpreters.

- **The profile is not the settings** PreferredColorScheme and DefaultDownloadFolderPath live on ICoreWebView2Profile, reached through ICoreWebView2_13::get_Profile at slot 105 - the same door _WV2_ClearBrowsingData already used, and a different object from the settings that _WV2_SettingRead and _WV2_SettingWrite walk. The settings are the behaviour of the page, the profile is the user of it, and the scheme a page sees belongs to the user. The profile is cached per view in $__WV2_pProf - one get_Profile call per view, not one per property read - and released after the loop that drops the versioned interfaces, which is step five of the teardown figure in chapter 7.
- **The ninth version in the cache** __WV2_ViewIf answers _2, _3, _4, _7, _8, _13, _16, _21 and _28 from one QueryInterface per handle and per rank. The controller of DefaultBackgroundColor is deliberately not in it: ICoreWebView2Controller2 is QI'd on the call and released on the way out, because a colour is not asked for often enough to be worth a cell in the heap.
- **A struct by value has no single shape** COREWEBVIEW2_COLOR is four bytes, so it crosses DllCallAddress as one integer with the alpha in the low byte; the RECT of put_Bounds is sixteen bytes, so x64 reads it by address. Both are true, and the difference is why the suite reads a colour back one channel at a time, with three different channels, twice, on each bitness - a swapped pair cannot pass that.
- **What the runtime refuses, and what the UDF refuses first** Only alpha 0 and 255 exist for a background colour; anything else is E_INVALIDARG from the runtime, so __WV2_BadByte and the alpha test turn it down before the call and the caller gets $WV2_ERR_ARG with no HRESULT to explain. A fourth scheme number is refused the same way. The download folder must be absolute, an empty string is refused because the runtime's own answer to no folder is the Downloads folder of the user rather than an empty one, and a folder that does not exist is not refused: the SDK makes it at the first download, and the suite proves the promise by looking for the bytes of a real download in the folder it named and for their absence in the folder of the user.
- **The title bar is not COM** _WV2_SetDarkTitleBar calls DwmSetWindowAttribute with attribute 20 - 19 on the builds before 19045 - and a BOOL passed by address in a four-byte struct. There is no getter, so the UDF reports whether the window accepted the attribute, and the suite prints the build number it asked on.

Lot 0.6.5 - what the browser is told before it exists

Twenty-one functions: nine properties of the environment, the user agent of one view, and the getters and the reset around them. The lot is a machinery question because of where the nine values live - in an object the UDF builds and the runtime reads, which is the opposite of every object this file described until now. The numbers are in the seventh suite: 51 checks, 0 failures, on both interpreters.

- **There is no factory** Parsing the export table of the installed EmbeddedBrowserWebView.dll yields nine functions: no CreateCoreWebView2EnvironmentWithOptions, no options factory, no DllGetClassObject. CoCreateInstance is therefore not a door for a script that ships nothing, and the create path this UDF has used since 0.6.2 takes the options object as its fourth argument - so the object is built here. __WV2_OptionsForCreate returns 0 when no _WV2_SetEnv* function was ever called, which is why the default branch is still the call of 0.6.4.
- **Six cells, and their widths come from the header** The base interface carries four pairs at slots 3 to 10, so its vtable is eleven pointers; _2, _3, _5, _6 and _8 are five each, and five is complete rather than truncated because the header gives each of them exactly one get/put pair. A second pair on any of them would be read past the cell. _4 and _7 are answered E_NOINTERFACE, and on 156 the runtime asks base, _2, base, _3, _5, _6, _4, a private interface, _8 and another private one, and starts the browser with what it got.
- **Nothing of it may be local** A DllStruct is freed when the function that made it returns, and the runtime reads the object long after __WV2_BuildOptions returned - so the six vtables, the six cells and the twenty-one callbacks are globals, and the object is built once per script. AddRef and Release count and never free, exactly as the handler sinks of chapter 2

do.

- **A string is allocated the way the callee frees it** A get of a LPWSTR out-parameter answers a CoTaskMemAlloc buffer, never a SysAllocString BSTR: the runtime calls CoTaskMemFree on what it is handed, and the mismatch is heap corruption some frames away. A property that was never set answers NULL, which is how the loader spells no value and what keeps an unset option from being heard as an empty one.
- **Two HRESULTs wrote this lot** An empty TargetCompatibleBrowserVersion is refused by the internal factory with 0x80070057 before any property is read, measured on both bitnesses, so the UDF fills the version of the DLL it is about to call, and the suite compares it with _WV2_GetBrowserVersion of the live view. The other is E_NOINTERFACE on ICoreWebView2Settings2, an interface introduced in Win32 1.0.864.35: on an older runtime the agent is reported unreachable with $WV2_ERR_CALL rather than pretending it was written.
- **Per view, or per process?** The agent is a property of the settings of one view, so the suite starts a second view with the same options: it shows the flag while the first keeps its override. That is the only check that tells the two apart, and it is the same argument the scheme of 0.6.4 made about the profile.

The kinds the queue carries

EVENT	WHAT _WV2_GETEVENT CARRIES
$WV2_EVENT_RESOURCE_REQUESTED (9)	uri; $iValue 1 answered, 0 let through, 2 the handler failed
$WV2_EVENT_FIND (10)	match count as $sData; $iValue the active index, -1 for none
$WV2_EVENT_DOWNLOAD_STATE (11)	uri; a $WV2_DL_* state
$WV2_EVENT_DOWNLOAD_PROGRESS (12)	"received/total"; the download index
$WV2_EVENT_SCRIPT_DIALOG (13)	the message; a $WV2_DIALOG_* kind
$WV2_EVENT_ACCELERATOR (14)	the virtual key; a $WV2_KEYEVENT_* kind
$WV2_EVENT_CONTEXT_MENU (15)	empty; the command id the handler answered with, -1 for none
$WV2_EVENT_FAVICON_CHANGED (16)	the address the page declares
$WV2_EVENT_DEVTOOLS (17)	the event parameters as JSON
$WV2_EVENT_PLAYING_AUDIO_CHANGED (18)	empty; 1 while the document makes sound
$WV2_EVENT_MUTED_CHANGED (19)	empty; the mute of the view, 0 or 1
$WV2_EVENT_FULLSCREEN_CHANGED (20)	empty; 1 while an element fills the view
$WV2_EVENT_PERMISSION_REQUESTED (21)	the origin that asked; a $WV2_PERM_* kind
$WV2_EVENT_CONTENT_LOADING (22)	the uri being shown; 1 when it is one of the runtime's own error pages
$WV2_EVENT_SOURCE_CHANGED (23)	the new uri; 1 for a new document, 0 for the same document that only moved
$WV2_EVENT_DOM_CONTENT_LOADED (24)	the uri; the navigation id, the number the NavigationCompleted of the same navigation is filed under
$WV2_EVENT_WINDOW_CLOSE (25)	the uri of the document that asked to close; the UDF closes nothing

WHAT DID NOT CHANGE

The ABI rules, the pump, the four-way error plumbing and the sixteen slots of the instance table are the ones 0.3 shipped. Every chapter that follows describes the machinery common to the four versions; where a chapter shows a slot number, it is the one the current source uses.

01

THE MINIMAL COM MODEL

The COM model in AutoIt

One pointer, one table of function pointers, and two ways to call them.

Everything in this UDF rests on one fact: a COM interface pointer is the address of a pointer to a table of function pointers. Calling a method is calling table[slot] with the interface pointer as the first argument.

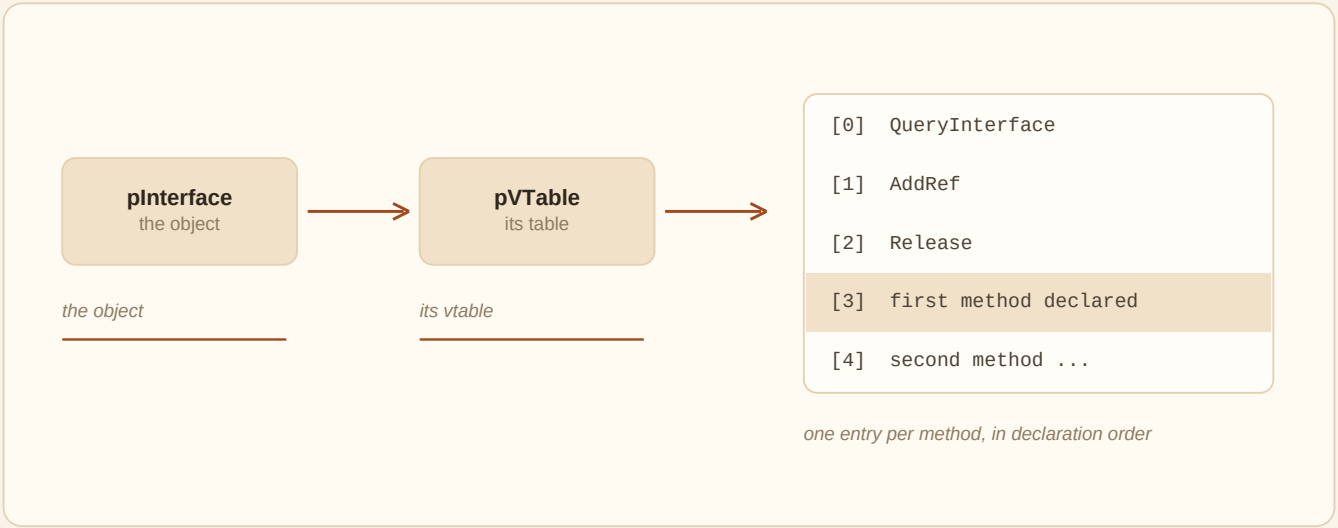


Figure 1. *Two dereferences: the object points at a table, the table points at code. Slot 3 is the first real method; slots 0 to 2 are always QueryInterface, AddRef and Release.*

Two ways to walk the table

ObjCreateInterface($p, $sIID, $sTag, $bCreate) builds a wrapper object whose methods you call with normal AutoIt syntax. The tag lists the methods in vtable order, starting at slot 0. That is the pleasant path, and it is what the UDF uses for the interfaces it speaks to constantly:

```
$__WV2_oCtl[$i] = ObjCreateInterface($pCtl, $WV2_IID_CTL, $__WV2_TAG_CTL, False)
; later, plain AutoIt:
Local $hr = $__WV2_oCtl[$i].put_ZoomFactor($fZoom)
```

The fourth argument is the one that matters most. With $bCreate = False the wrapper does not add a reference and does not own the pointer: it is a view of a reference somebody else holds. Every pointer the runtime hands to a callback is borrowed in exactly that way, which is why every wrapper here is built with False.

DllCallAddress("long", $pFn, "ptr", $pObj, ...) calls a raw slot, with $pFn read out of the table:

```
Func __WV2_VtFn($pObj, $iSlot)
    If $pObj = 0 Or $iSlot < 0 Then Return 0
    If $__WV2_iPtrSize = 0 Then
        $__WV2_iPtrSize = DllStructGetSize(DllStructCreate("ptr"))
    Local $pVTable = DllStructGetData(DllStructCreate("ptr", $pObj), 1)
    If @Error Or $pVTable = 0 Then Return 0
    Local $pFn = DllStructGetData( _
            DllStructCreate("ptr", $pVTable + $iSlot * $__WV2_iPtrSize), 1)
    If @Error Or $pFn = 0 Then Return 0
    Return $pFn
EndFunc
```

Reading a vtable is three lines: dereference the object, add slot times sizeof(pointer), read the entry. The UDF uses it for two jobs: methods that exist only on a newer version of an interface (chapters 3 and 5), and Release on pointers owned by the operating system, because wrapping those with ObjCreateInterface crashed the process in this build.

The environment is not created through COM

There is no CoCreateInstance in this UDF. The environment is built by a plain export of the runtime's own DLL, called through DllCall:

```
Local $aCE = DllCall($sDll, "long", _
        "CreateWebViewEnvironmentWithOptionsInternal", _
        "bool", 1, "int", 0, "wstr", $sUserDataFolder, "ptr", 0, _
        "ptr", $__WV2_pObj[$i][$__WV2_H_ENV])
```

Five arguments: should the runtime create a feedback handler, an options word, the user data folder, a reserved pointer, and the address of the callback object that will receive the environment. This is the call the official WebView2Loader.dll makes on your behalf. Making it directly is what removes the last file from the distribution - and it is the one undocumented thing in the package. Only this line depends on it.

Anatomy of _WV2_Create

Seven steps, in this order, and the order is not negotiable:

1. **Parent window** Validate it, then take a free slot (1 to 16).
2. **Runtime** Locate it: the registry key Microsoft\EdgeUpdate\Clients\{F3017226-...}, then EdgeWebView\Application. Bitness matters: a 32-bit AutoIt loads the 32-bit DLL.
3. **Apartment** CoInitializeEx(NULL, COINIT_APARTMENTTHREADED), once per script. The UDF never calls CoUninitialize, so a COM-aware script keeps working after a view is destroyed.
4. **Handlers** Build the thirty-nine handler objects of the slot (chapter 2).
5. **Environment** Call CreateWebViewEnvironmentWithOptionsInternal and pump until the environment callback answers, with a 60 s default timeout.
6. **Controller** The environment callback immediately asks for a controller, which arrives as a second callback; the controller is wrapped, put_Bounds and put_IsVisible(1) are sent.
7. **Ready** Subscribe NavigationCompleted, read the browser version, mark the handle ready.

If any step fails, __WV2_Fail tears down whatever exists, the slot returns to the table, and _WV2_Create returns 0 with @error and @extended set. Nothing is left half-alive.

02

FAKE VTABLES IN MEMORY

Handler objects in a global heap

The runtime does not pull events: it calls back. AutoIt has to grow an object.

The runtime calls an ICoreWebView2XxxEventHandler whose only real method is Invoke. AutoIt cannot create a COM object, so the UDF manufactures one out of memory: one block of five pointers per instance and per slot.



Figure 2. *The object is the last cell; its one field points back at the first, which is the vtable. That is exactly the layout COM expects, built with DllStructSetData.*

The object address is block + 4 pointers. Its first field points at the vtable that begins at block, which is exactly what COM expects. __WV2_BuildHandler writes the five entries:

```
Func __WV2_BuildHandler($i, $iSlot)
    Local $b = __WV2_BlockBase($i, $iSlot)
    DllStructSetData($__WV2_tHeap, "data", DllCallbackGetPtr($__WV2_hCbQI), $b + 1)
    DllStructSetData($__WV2_tHeap, "data", DllCallbackGetPtr($__WV2_hCbAddRef), $b + 2)
    DllStructSetData($__WV2_tHeap, "data", DllCallbackGetPtr($__WV2_hCbRelease), $b + 3)
    DllStructSetData($__WV2_tHeap, "data", _
            DllCallbackGetPtr($__WV2_hCbInv[$iSlot]), $b + 4)
    Local $pBase = DllStructGetPtr($__WV2_tHeap)
    DllStructSetData($__WV2_tHeap, "data", $pBase + $b * $__WV2_iPtrSize, $b + 5)
    $__WV2_pObj[$i][$iSlot] = $pBase + ($b + 4) * $__WV2_iPtrSize
EndFunc
```

WHY A GLOBAL STRUCT

One global DllStruct owns all the memory: 1120 pointers, allocated once on first use and never resized. A global struct has no copy semantics, so the addresses stay valid for the life of the script - a callback running while AutoIt reallocated a local array would be a use-after-free waiting to happen.

One callback set, thirty-nine entry points

There is no per-instance AutoIt function. There is one QueryInterface, one AddRef, one Release and one Invoke per slot, registered once with DllCallbackRegister. The Invoke receives $pThis, and __WV2_FindInst maps that pointer back to (instance, slot):

```
Func __WV2_FindInst($pThis)
    Local $i, $j
    For $i = 1 To $__WV2_MAX
        If $__WV2_aUsed[$i] Then
            For $j = 1 To $__WV2_SLOT_COUNT
                If $__WV2_pObj[$i][$j] = $pThis Then Return SetError(0, $j, $i)
            Next
        EndIf
    Next
    Return SetError(1, 0, 0)
EndFunc
```

A linear scan of at most 496 comparisons, against a callback that fires on a user action: the cost is irrelevant, and it buys a completely uniform callback set. It also lets the Invoke signatures differ per slot, because the argument list is registered from $__WV2_aInvArgs: a two-argument completion handler (this, hr) and a three-argument event handler (this, sender, args) live in the same block.

QueryInterface is a gate, not a formality

Each slot answers for IUnknown and for the handler interface of its own slot, and nothing else. Claiming an interface whose vtable you do not carry is the fastest way to kill the process: the runtime would call Invoke at a slot your object does not have.

```
If $bKnown And ($sIID = $WV2_IID_UNKNOWN _
        Or $sIID = $__WV2_aInvIID[$iSlot]) Then $pOut = $pThis
DllStructSetData(DllStructCreate("ptr", $pPPV), 1, $pOut)
If $pOut = 0 Then Return $__WV2_HR_E_NOINTERFACE
Return 0
```

AddRef returns 1 and Release returns 0, both without touching memory. These objects live in the UDF heap and are freed by __WV2_FreeHandlers, never by the runtime. The counters are deliberately meaningless: a fake object that released itself would dangle the moment the runtime kept a pointer. __WV2_FindInst failing is a normal outcome - a late callback after teardown - and every Invoke returns early in that case.

Subscribing: the token is the receipt

add_Xxx(handler, ByRef token) returns an EventRegistrationToken (int64) that remove_Xxx needs. __WV2_Attach and __WV2_Detach keep one token per instance and slot and refuse to subscribe twice. The method name cannot be built at run time in AutoIt, so each slot gets its own Case: a little repetition standing in for a dispatch mechanism AutoIt does not have. NavigationCompleted is subscribed by _WV2_Create itself, because _WV2_WaitNavigation waits on it.

NEXT CHAPTER 03 Vtable ranks and WebView2.h

03

COUNTED FROM THE HEADER

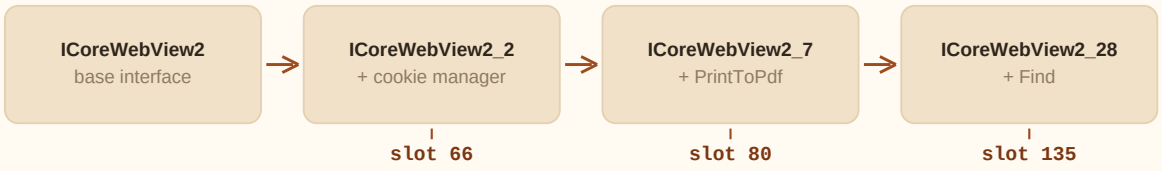
Vtable ranks and WebView2.h

Slot 3 is the first real method; everything after it is a count.

The numbering rule

Slot 0 is QueryInterface, 1 is AddRef, 2 is Release. The first method an interface declares is at slot 3. A versioned interface inherits the whole chain: its first own method sits right after the last method of every earlier version.

```
slot(method) = 3 + number of methods declared before it in the chain
```



The chain is counted, never guessed: the versions in between are _3, _4, _11, _13, _15, _16 and _21.

Figure 3. *Four of the ranks this UDF walks, from the base interface to the one that carries the search object.*

That count must be taken from the official header - never from memory, never from a summary page. The SDK's WebView2.h is the only authoritative order.

The slots this UDF calls raw

INTERFACE	SLOT	WHAT THE RAW CALL BUYS
IUnknown	0, 2	QueryInterface and Release on a pointer the system owns
IStream	3, 5	Read and Seek: the PDF of _WV2_PrintToPdfStream, the bytes of a favicon
ICoreWebView2Controller	6	put_Bounds - the ABI trap of chapter 4
ICoreWebView2Controller2	26, 27	get_ and put_DefaultBackgroundColor, the colour behind a page that has not painted
ICoreWebView2Controller3	19, 20	add_ and remove_AcceleratorKeyPressed
ICoreWebView2	36, 42	CallDevToolsProtocolMethod and its event receiver
ICoreWebView2	52 to 54	the full screen: add_ and remove_ContainsFullScreenElementChanged, get_ContainsFullScreenElement
ICoreWebView2	55 to 58	the resource layer: the two event slots and the two filter slots
ICoreWebView2_2	66	get_CookieManager, the door to every cookie call

INTERFACE	SLOT	WHAT THE RAW CALL BUYS
ICoreWebView2_3	71, 72	the folder served under a real host name, and its Clear
ICoreWebView2_4	75, 76	add_ and remove_DownloadStarting
ICoreWebView2_7	80	PrintToPdf
ICoreWebView2_8	81 to 87	the sound: add_ and remove_IsMutedChanged, get_ and put_IsMuted, the two audio event slots and get_IsDocumentPlayingAudio
ICoreWebView2_11	100, 101	add_ and remove_ContextMenuRequested
ICoreWebView2_13	105	get_Profile, the door both _WV2_ClearBrowsingData and the theme calls go through
ICoreWebView2Profile	6 to 9	the two strings and the two numbers of the profile: the download folder, the scheme the page sees
ICoreWebView2_15	109 to 112	the favicon: two event slots, get_FaviconUri, GetFavicon
ICoreWebView2_16	113 to 115	Print, ShowPrintUI, PrintToPdfStream
ICoreWebView2_21	122	ExecuteScriptWithResult, the typed answer
ICoreWebView2_28	135	get_Find
ICoreWebView2Environment6 / _15	14 / 29	CreatePrintSettings and CreateFindOptions
ICoreWebView2Find	3 to 12	the two counters, their events, Start, FindNext, FindPrevious, Stop
ICoreWebView2CookieManager	3 to 10	CreateCookie, GetCookies, AddOrUpdateCookie, DeleteAllCookies
ICoreWebView2Cookie	3 to 16	name, value, domain, path, expiry, SameSite, HttpOnly, Secure
The resource and download args	3 to 9	the uri, the method, the headers, the answer put back
The permission args and their deferral	3 to 8 / 3	uri, kind, get_ and put_State, GetDeferral - and Complete, the only way to answer after the handler returned
ICoreWebView2EnvironmentOptions	3 to 10	the four scalar properties of the environment - and the object is the UDF's, so the runtime is the caller (lot 0.6.5)
Options _2, _3, _5, _6, _8	3, 4	one pair each, which is why those five vtables are five pointers wide and no wider
ICoreWebView2Settings2	21, 22	get_ and put_UserAgent of one view

That table is a digest, not the list. The list is two hundred and eleven named constants, each one spelling its own interface and method - $__WV2_SLOT_ICoreWebView2_16_Print - and Tools/gen_interfaces.py --audit recounts every one of them, the interface identifiers with them, against the header: 300 constants, zero differences on SDK 1.0.4314.35. The 0.3 edition of this chapter listed thirteen raw calls; the growth of that number is what the versions after it cost, and the audit is what makes it payable.

Every number was counted from the header, then confirmed at run time by calling the slot and checking the HRESULT - and, where the method has an observable effect, by looking at the effect (chapter 4).

A tag may stop early; it may not skip

$__WV2_TAG_WV stops at get_DocumentTitle. That is legal: a tag describes the methods you intend to call, and a wrapper only knows the ones you declared. What is not legal is a tag that skips a method in the middle: the wrapper counts positions, so a missing entry shifts everything after it by one and the call lands on the wrong function.

That is the whole reason PrintToPdf, at slot 80, is not in a tag. A complete ICoreWebView2_7 tag would carry hundreds of lines for one call, and the wrapper would gain nothing: DllCallAddress needs the same arguments and one constant.

QI is the version check

Asking for ICoreWebView2_7 on a runtime older than that interface returns E_NOINTERFACE (0x80004002) - not a crash, not a silent failure. __WV2_QiView converts the textual IID with CLSIDFromString, calls slot 0 on the raw view pointer, and keeps the answer; __WV2_ViewIf asks once per version per instance, and the pointers are released with the view. The distinction it makes matters:

- **E_NOINTERFACE** becomes $WV2_ERR_CALL with that HRESULT: the machine's runtime is older than the feature. The caller degrades; the UDF does not lie about it.
- **Any other failure** becomes a COM error: the UDF's own bookkeeping is wrong.

04

THE ABI TRAP

x86 and x64: put_Bounds

The header says by value. The 64-bit runtime reads a pointer. One function carries the difference.

ICoreWebView2Controller::put_Bounds takes a RECT by value. On x86 (stdcall) the four longs are pushed on the stack, and the wrapper tag put_Bounds hresult(long;long;long;long) describes exactly that. It works.

On x64 the first four arguments travel in RCX, RDX, R8 and R9, and a structure that does not fit in eight bytes is never passed by value: the caller passes its address. The header still says by value; the wire says pointer. Give the callee four longs and it reads left as a pointer and dereferences it - that is the crash.

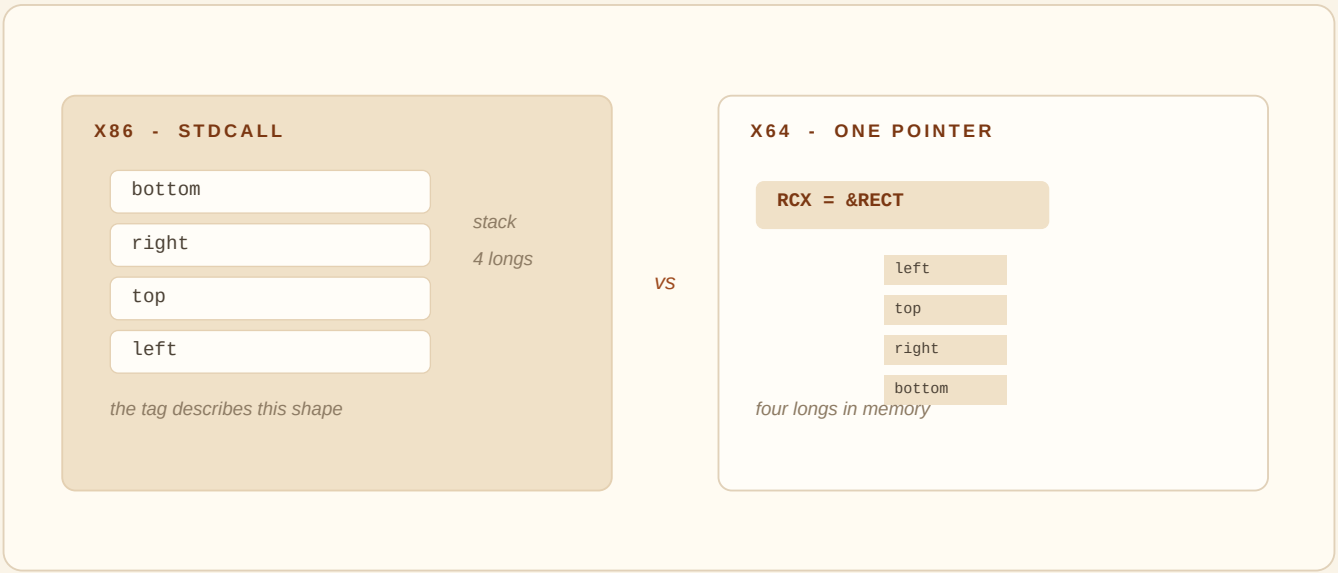


Figure 4. *The same declaration, two wire formats. Giving the callee four longs on x64 makes it dereference the first one as an address: the process terminates.*

What was measured

Four shapes were tried on the 64-bit runtime, in a scratch script that logged each step to a file, because a crash truncates the console but not a flushed file:

ARGUMENT SHAPE	RESULT
Four long in a tag (the correct x86 shape)	process terminates
Two int64 (the sixteen bytes in two registers)	process terminates
AutoIt's by-value struct	process terminates
ptr to a DllStruct of four long, through the vtable	hr = 0, bounds applied

The fourth becomes the only 64-bit path:

```
Func __WV2_PutBounds($i, $iLeft, $iTop, $iRight, $iBottom)
    If Not @AutoItX64 Then
        Return __WV2_Result( _
            $__WV2_oCtl[$i].put_Bounds($iLeft, $iTop, $iRight, $iBottom), _
            "ICoreWebView2Controller::put_Bounds")

    Local $pFn = __WV2_VtFn($__WV2_pCtl[$i], $__WV2_SLOT_ICoreWebView2Controller_put_Bounds)
    If $pFn = 0 Then
        __WV2_SetErr($WV2_ERR_COM, 0, "__WV2_VtFn")
        Return 0
    EndIf
    Local $tRc = DllStructCreate("long l; long t; long r; long b")
    DllStructSetData($tRc, "l", $iLeft)
    DllStructSetData($tRc, "t", $iTop)
    DllStructSetData($tRc, "r", $iRight)
    DllStructSetData($tRc, "b", $iBottom)
    Local $aCall = DllCallAddress("long", $pFn, "ptr", $__WV2_pCtl[$i], _
            "ptr", DllStructGetPtr($tRc))
    If @Error Or Not IsArray($aCall) Then
        __WV2_SetErr($WV2_ERR_COM, 0, "ICoreWebView2Controller::put_Bounds")
        Return 0
    EndIf
    Return __WV2_Result($aCall[0], "ICoreWebView2Controller::put_Bounds")
EndFunc
```

No single tag can be correct on both bitnesses here, which is why the function branches instead of carrying one declaration.

A struct by value has no single shape

put_Bounds is the reason this guide says a struct passed by value travels as an address on x64. DefaultBackgroundColor, the property lot 0.6.4 added, is the reason the same guide has to say the opposite as well: COREWEBVIEW2_COLOR is {BYTE A, R, G, B} - four bytes, which is one register - so it crosses DllCallAddress as a single 32-bit integer with the alpha in the low byte, the same shape on both bitnesses and no branch at all:

BY-VALUE STRUCT	HOW IT CROSSES	WHAT THE UDF DOES
RECT, 16 bytes (put_Bounds)	x86 four longs on the stack; x64 the address of the struct	branches on @AutoItX64
COREWEBVIEW2_COLOR, 4 bytes	one 32-bit integer, alpha in the low byte, both bitnesses	BitOR and three BitShift, no branch

Treating the two the same either crashes or quietly swaps channels, and the swap is the quieter of the two: __WV2_PackColor builds the integer, __WV2_ReadColor takes the four bytes back one field at a time, and the suite writes 30/60/90 then 12/200/77 and checks every channel on each interpreter. One more shape from the same lot belongs to no interface at all: the BOOL _WV2_SetDarkTitleBar hands to DwmSetWindowAttribute is passed by address, in a four-byte struct, because that API wants a pointer to a BOOL - the third convention in three calls.

hr = 0 is not proof

The first version of the 64-bit fix returned S_OK and did nothing visible. S_OK only means the call arrived; it says nothing about what the callee did with the arguments. The fix was accepted only after three independent proofs:

- **Pixel** GetPixel on the parent window: inside the browser area the pixel is the page colour, outside it is the GUI background.
- **Resize** The boundary moved when the window was resized and _WV2_SetBounds was called again.

- **Read back** get_Bounds returned exactly the rectangle that had been sent.

Apply that rule to every raw call you add: find an observable consequence and check it.

The other shapes you will meet

VALUE	AUTOIT TYPE	NOTE
HRESULT	long	negative failure values; @extended keeps it
BOOL	int	32-bit, never bool (one byte)
long / DWORD	long	32-bit on both bitnesses (LLP64 on x64)
EventRegistrationToken	int64	by value on remove_*, by pointer on add_*
LPWSTR returned	wstr*	allocated with CoTaskMemAlloc
interface out-parameter	ptr*	carries a reference of its own

A text returned by the runtime is copied and freed by __WV2_WStrProp. A returned interface pointer is released with __WV2_ReleaseExternal unless it is wrapped and becomes the UDF's property.

05

A WORKED EXAMPLE

Adding a method in six steps

Worked example: the two calls behind _WV2_ClearCookies.

get_CookieManager lives on ICoreWebView2_2 at slot 66; DeleteAllCookies lives on ICoreWebView2CookieManager at slot 10. Neither is in a tag, and the six steps that brought them into the UDF are the same for any other method.

1. Find the declaration

Open WebView2.h from the SDK and locate the interface that declares the method. Note the exact parameter list and the return type of every argument. A method found on a documentation page is a hint; the header is the fact.

2. Turn the position into a slot

Count every method declared before it in the whole inheritance chain, add 3. Record the number as a named constant next to the others, with the interface name in the comment: a bare 66 in the middle of a function is unreviewable.

```
Global Const $__WV2_SLOT_ICoreWebView2_2_get_CookieManager = 66
Global Const $__WV2_SLOT_ICoreWebView2CookieManager_DeleteAllCookies = 10
```

3. Ask the view for that interface version

```
Local $p2 = __WV2_ViewIf($iHandle, $__WV2_IF_2)
If $p2 = 0 Then Return SetError($__WV2_iLastError, $__WV2_iLastHR, 0)
```

If $p2 is 0 because of E_NOINTERFACE, the runtime is older than the feature. __WV2_SetErr has already recorded that, and the public function propagates it unchanged.

4. Write the call shape

Four helpers cover most methods; anything else goes through DllCallAddress directly.

HELPER	SIGNATURE IT CALLS
__WV2_CallNoArg($pObj, $slot)	hr Method()
__WV2_CallBool($pObj, $slot, $v)	hr Method(BOOL)
__WV2_CallWStr($pObj, $slot, $s)	hr Method(LPCWSTR)
__WV2_WStrProp($pObj, $slot)	hr get_X(LPWSTR*) - returns the text

get_CookieManager is none of these: it returns an interface pointer, so it is written out. Three checks follow every raw call - @error, the HRESULT, and a non-null pointer:

```
Local $tOut = DllStructCreate("ptr")
Local $aCall = DllCallAddress("long", $pFn, "ptr", $p2, _
        "ptr", DllStructGetPtr($tOut))
If @Error Or Not IsArray($aCall) Then
    __WV2_SetErr($WV2_ERR_COM, 0, "ICoreWebView2_2::get_CookieManager")
    Return SetError($__WV2_iLastError, $__WV2_iLastHR, 0)
EndIf
If $aCall[0] <> 0 Then
    __WV2_SetErr($WV2_ERR_CALL, $aCall[0], "ICoreWebView2_2::get_CookieManager")
    Return SetError($__WV2_iLastError, $__WV2_iLastHR, 0)
EndIf
Local $pCookie = DllStructGetData($tOut, 1)
```

5. Own what comes back

An interface returned through an out-parameter carries its own reference; QueryInterface and every getter that hands out an object do this. Either wrap it and release it with the rest of the instance, or release it as soon as you are done:

```
Local $hr = __WV2_CallNoArg($pCookie, $__WV2_SLOT_ICoreWebView2CookieManager_DeleteAllCookies)
__WV2_ReleaseExternal($pCookie)
```

Text is different: it is CoTaskMemAlloc'd, so it is copied (__WV2_CopyWStr) and then freed with CoTaskMemFree - never handed to the caller as-is. Since 0.6.5 the UDF also stands on the producing side of that rule: the options object it hands the runtime implements getters that answer a LPWSTR*, and there the buffer must be made with CoTaskMemAlloc and never a SysAllocString BSTR, because the caller frees with CoTaskMemFree.

6. Wrap it, then prove it

```
Func _WV2_ClearCookies($iHandle)
    $__WV2_sLastFunc = "_WV2_ClearCookies"
    If Not __WV2_Guard($iHandle) Then
        Return SetError($__WV2_iLastError, $__WV2_iLastHR, 0)
    ...
    Local $bOk = __WV2_Result($hr, _
            "ICoreWebView2CookieManager::DeleteAllCookies")
    Return SetError($__WV2_iLastError, $__WV2_iLastHR, $bOk)
EndFunc
```

Then add a check to WebView2_Example_GUI.au3 (--auto) that exercises the new call and asserts its effect, not its return value. The suite is the specification of what must keep working; a feature without a check there is a feature that will silently break.

When to extend a tag instead

If the interface is fully declared in the tag up to the method you need, and you expect to call several of its methods, extending the tag is cheaper and type-checked by AutoIt's own argument conversion. The rule of thumb: one call, raw slot; a family of calls, tag.

NEXT CHAPTER 06 Diagnosing a failure

06

TWO INTEGERS AND A LABEL

Diagnosing a failure

Two integers and a label; a crash that prints nothing; the calls that answer later.

The error plumbing

Every failure goes through one function: __WV2_SetErr stores the category, the raw HRESULT and the label. The public function returns SetError($__WV2_iLastError, $__WV2_iLastHR, value), so @error carries the category and @extended the HRESULT. _WV2_GetLastErrorMessage turns both into one line:

```
Method call failed in ICoreWebView2_7::PrintToPdf (HRESULT 0x80070005)
```

Three habits make that usable:

- **Never Return f().** SetError does not survive a plain Return; every public function sets @error and @extended explicitly. This was a real bug in an earlier version, invisible on the success paths.
- **Label the call site** ICoreWebView2_7::PrintToPdf tells you which method failed; PrintToPdf does not.
- **No output** The UDF never prints and never shows a dialog. A script can run in a service or under a scheduler with no console. Diagnostics are values, not output.

Reading an HRESULT

VALUE	NAME	USUAL CAUSE HERE
0x80004001	E_NOTIMPL	the runtime does not implement this method
0x80004002	E_NOINTERFACE	runtime older than the interface version
0x80004005	E_FAIL	the call was made, the runtime refused
0x80070002	ERROR_FILE_NOT_FOUND	a path handed to the runtime does not exist
0x80070005	ERROR_ACCESS_DENIED	profile or output folder not writable
0x80070057	E_INVALIDARG	wrong argument shape - suspect the ABI

A crash that prints nothing

A wrong slot or a wrong argument shape does not return an HRESULT: the process disappears, and the console loses whatever was buffered. Bisect like this:

- **Flush** Write a scratch script with ConsoleWrite and FileWriteLine on a handle opened with $FO_APPEND, flushed after each line.
- **Last call** Call the suspect method last, with everything else already proven.
- **Remove it** Comment out the call: if the script completes, the call is the killer.
- **One variable** Vary one thing at a time - the slot, then each argument type, then the order of the returned value and the release.
- **Prove it** When a shape succeeds, do not stop: prove the effect.

Two less obvious traps. A wrong-arity call to your own AutoIt function is a runtime fatal, not a compile error:

Au3Check passes and the script exits 1 with a truncated log, so count the arguments before suspecting COM. And ObjCreateInterface on a pointer the operating system owns crashed the process in this build, while reading Release from the vtable does not: when a wrapper is not yours, do not wrap it.

Asynchronous methods

ExecuteScript, AddScriptToExecuteOnDocumentCreated, CapturePreview and PrintToPdf answer through a handler, later. The pattern is the same for every waiting call - the 0.5 lot added ExecuteScriptWithResult, GetCookies, ClearBrowsingData, PrintToPdfStream, Start, GetFavicon and CallDevToolsProtocolMethod to that family, each with a slot of its own in the $__WV2_H_* table:

```
__WV2_BeginOp($iHandle, $__WV2_H_PRINT)       ; clear the state
Local $aCall = DllCallAddress(...)             ; start
__WV2_WaitOp($iHandle, $__WV2_H_PRINT, $iMs)   ; pump until it answers
Local $iDoneHR = $__WV2_iOpHR[$iHandle][$__WV2_H_PRINT]
```

__WV2_WaitOp pumps the message queue in 20 ms slices, so the GUI keeps repainting while the script waits. Three distinct failures must stay distinguishable, and _WV2_PrintToPdf shows all three: the call itself fails (WV2_ERR_CALL with its HRESULT), the handler reports an HRESULT (WV2_ERR_CALL with that HRESULT), or the call succeeds and the handler answers FALSE (WV2_ERR_FILE, because a PDF that was not written is not a method failure).

The event queue

Events are pushed into a 64-entry ring per instance and dropped oldest-first when the script does not drain them; $__WV2_iEvtDrop keeps the count. _WV2_GetEvent on an empty queue returns 0 with @error = 0: empty is not an error, and a loop must tell the two apart. _WV2_WaitEvent waits for one specific type and leaves the others in the queue.

NEXT CHAPTER 07 Threading and the message pump

07

ONE APARTMENT, ONE PUMP

Threading and the message pump

One apartment, one thread, and nothing happens without a pump.

One apartment, one thread

CoInitializeEx(NULL, COINIT_APARTMENTTHREADED) runs once per script, on the thread that called _WV2_Create. The browser is created in that apartment and delivers every event on that thread, through the Windows message queue. Consequences:

- **One thread** A view belongs to the thread that created it. Calling it from another thread is undefined behaviour; no marshalling is done.
- **No pump, no events** Nothing arrives until messages are dispatched: no pump, no events, no completion handlers, no navigation progress.
- **Waiting still paints** _WV2_Create and every waiter pump while they wait, which is why a script can block on a page load and still repaint.

The pump

```
Func __WV2_Pump($iMs)
    Local Static $tMsg = DllStructCreate( _
        "ptr hwnd;uint message;ptr wParam;ptr lParam;dword time;long x;long y")
    Local $tEnd = TimerInit()
    While TimerDiff($tEnd) < $iMs
        Local $aPM = DllCall("user32.dll", "bool", "PeekMessageW", _
            "ptr", DllStructGetPtr($tMsg), "ptr", 0, _
            "dword", 0, "dword", 0, "dword", 1)
        If @Error Or Not $aPM[0] Then
            Sleep(2)
        Else
            DllCall("user32.dll", "bool", "TranslateMessage", _
                "ptr", DllStructGetPtr($tMsg))
            DllCall("user32.dll", "lresult", "DispatchMessageW", _
                "ptr", DllStructGetPtr($tMsg))
        EndIf
    WEnd
EndFunc
```

PM_REMOVE (1), every message, no filtering: the runtime posts its work as ordinary window messages, and filtering them out would starve it. Sleep(2) when the queue is empty keeps a waiter from spinning a core.

Inside a GUI loop

The pattern that works: do not wait inside GUIGetMsg handlers. Either wait before entering the loop, or let the main loop pump - it already does - and read events with _WV2_GetEvent. A long _WV2_WaitNavigation inside a button handler freezes the toolbar for as long as the page takes: correctly, visibly, and avoidably.

Teardown order

_WV2_Destroy runs __WV2_Teardown: thirteen releases in a fixed order, because an interface must outlive every pointer taken from it. The versioned interfaces, the search object, the profile, the CDP receiver, the remembered downloads and the print settings all point into the same runtime object as the view, so they go first; the environment, which created the controller, goes last. Two adaptive drains, __WV2_PumpDrain(100, 20), separate the controller from the environment, so the runtime can retire its own references before the last one goes - and return after twenty milliseconds of silence instead of on a timer.



An adaptive drain returns after 20 ms of silence and never runs past 100 ms, so an idle close does not pay for a timer it has no use for.

Figure 5. *The thirteen steps of __WV2_Teardown, read left to right. The profile is fifth because it was taken from ICoreWebView2_13, which the loop at three has already released. Inverting the second and the eleventh leaves the runtime holding an interface into freed memory - and the crash shows up at exit, far from the cause.*

What a close does not do

Two things were taken out of this path, both because of what the user feels. The temporary profile folder is not deleted here: the browser process holds it for seconds after the controller closes, so a delete on the way out is fifteen seconds of walking a tree it cannot finish - measured, 15 542 ms, with the window frozen on screen. __WV2_Free queues the folder instead, and __WV2_SweepStale collects it, with the rubbish of the runs that were killed, at the end of the next _WV2_Create. Only a folder the UDF built itself is ever queued or swept ($__WV2_bTempUser, and __WV2_IsOurProfile), and never one whose process is still alive.

And nothing is registered with OnAutoItExitRegister. An exit function runs after the message loop is gone, which is exactly where a teardown needs the loop: the same work, one view, the window closed from outside, measured from the click to the moment the process stops existing.

WHAT THE SCRIPT DOES WHEN THE USER CLOSES	WINDOW OFF SCREEN	PROCESS GONE
Destroy in the message loop	252 - 376 ms	295 - 521 ms
nothing at all (the script forgot)	254 - 275 ms	286 - 1 935 ms
exit handler: Close on the controller	252 - 254 ms	664 - 680 ms
exit handler: release controller and environment	256 - 257 ms	393 - 827 ms

WHAT THE SCRIPT DOES WHEN THE USER CLOSES	WINDOW OFF SCREEN	PROCESS GONE
exit handler: full teardown, pumps left in	251 - 263 ms	1 705 - 2 702 ms
exit handler: full teardown, pumps taken out	260 - 261 ms	1 387 - 2 109 ms

A net placed in an exit handler would make closing three to six times slower and would protect nothing: the folders a forgotten destroy leaves behind carry the pid that made them, the next create recognises them as rubbish, and the browser processes leave on their own. The net the UDF does offer is _WV2_DestroyAll - every live view, one call, in the loop where the runtime can still answer.

DESTROY IS REFUSED INSIDE A CALLBACK

A resource, dialog, key or menu handler is walking the runtime's own COM objects while it runs. Releasing them from there - or pumping messages, which would deliver the next request inside this one - would deadlock. _WV2_Destroy checks the two re-entrancy flags of the instance and answers $WV2_ERR_BUSY instead; set a flag and tear the view down from the main loop.

The browser is a window, not a control

_WV2_Create($hParent, ...) puts a real child window inside $hParent. It covers every AutoIt control whose rectangle overlaps its own, and GUICtrlSetResizing does not apply - it has no control identifier. Either reserve a band of the client area for the toolbar, or call _WV2_FitToParent or _WV2_SetBounds from the resize handler.

NEXT CHAPTER 08 Limits, pitfalls, references

08

WHAT IT COSTS, WHERE TO LOOK

Limits, pitfalls, references

What the wrapper bounds, what the runtime decides, and where to read more.

Limits of the wrapper

LIMIT	VALUE	WHERE IT COMES FROM
Concurrent views	16	$__WV2_MAX
Handler slots per view	39	$__WV2_SLOT_COUNT
Pointer cells in the heap	3 120	16 x 39 x 5, one DllStruct
Queued events per view	64	$__WV2_QCAP, oldest dropped
Downloads remembered per view	8	$WV2_DL_MAX
Permission requests held per view	8	$WV2_PERM_MAX; a ninth takes the slot of the oldest, unanswered
Permission kinds with a policy	14	$WV2_PERM_UNKNOWN to $WV2_PERM_PERSISTENT_STORAGE
Profile folders waiting for the sweep	64	$__WV2_MAX x 4, released by the next create
Live print settings objects	8	$WV2_PRINT_SETTINGS_MAX
Create / navigation timeout	60 s / 30 s	$WV2_CREATE_TIMEOUT, $WV2_NAV_TIMEOUT
Script, find / print timeout	5 s / 30 s	$WV2_EXEC_TIMEOUT, $WV2_PRINT_TIMEOUT
Zoom range	0.2 to 10	validated by _WV2_SetZoom

Pitfalls worth knowing before they cost a day

- **Undocumented entry point** The loaderless call is stable in practice and confined to one DllCall in _WV2_Create. If a future runtime ever moved it, the fallback is to ship WebView2Loader.dll and change that one call - nothing else touches it.
- **A profile outlives the view** The browser process holds the folder for a few seconds after _WV2_Destroy, which cannot always delete it; the next _WV2_Create on the same slot does. Pass your own folder when the profile must survive or be audited.
- **Cookies need an origin** A file:// page or NavigateToString has a null origin and cannot store a cookie: document.cookie stays empty by design, and that is not a bug in the bridge. The virtual host mapping described on the previous page gives your own content a real origin.
- **Four kinds, not five** _WV2_ClearCookies wipes the jar. _WV2_ClearBrowsingData takes history, downloads, cookies and cached files - the four kinds of ICoreWebView2Profile2 - and localStorage is not one of them. Clear it from the page.
- **One download per page** Chromium allows one automatic download per page. A second scripted download is dropped silently and raises no event. A redirected download reaches the disk 0 to 3 s after DownloadStarting: wait for the file instead of concluding it failed.
- **A blob download never finishes** A download made from a blob URL stays in the running state while the page that

made it lives, and the size it reports stays unknown. Poll the file, not the state, and let the page go before expecting the transfer to close.

- **Filter needs the event** _WV2_SetNavigationFilter needs $WV2_EVENT_NAVIGATION_STARTING enabled. The filter is applied by that handler; without the subscription there is nothing to cancel.
- **ExecuteScript returns JSON** document.title comes back as "Hello" with quotes, a number as 42, and a script that throws as null with @error = 0. Test for "null" rather than for failure, or call _WV2_ExecuteScriptEx, which reports the exception text.
- **The first search is the slow one** The search object is built on the first _WV2_FindStart of a view, and that one call can exceed the 5 s default while the engine wakes up. Pass a longer timeout for the first search, or run it once when the page loads.
- **Destroy is refused inside a handler** _WV2_Destroy answers $WV2_ERR_BUSY while a resource, dialog, key or menu handler of that view is running. Set a flag and destroy from the main loop - chapter 7 explains why re-entering the runtime from there deadlocks it.
- **BOOL is four bytes** bool in a DllCall is one byte and corrupts the stack on the way back. Every BOOL in this UDF is int.
- **The folder moves, the numbering does not** put_ResultFilePath moves the folder, not the name: the runtime has already picked the file name where it would have written by itself, the Downloads folder of the user, and numbered a name it found used there. A page offering report.pdf can therefore land as report (3).pdf in an empty folder of your own. Section 21 of the self test names its payload after the process for exactly that reason.

Testing

WebView2_Example_GUI.au3 --auto runs 297 checks over the whole API in 37 sections without network access (file://, NavigateToString and a virtual host). It writes only in @TempDir, in folders named after the process that made them: the profile folders its views release are taken back by the sweep of chapter 7, and the two scratch folders of sections 21 and 36 are removed at the end of the run - a browser process that has not let go of the second one is reported, not failed on. Read the first line of section 2 before you trust a failure: when the machine says it is loaded, the four duration ceilings report instead of judging, and a FAIL on one of them is the machine's memory, not your change - the property checks never bend. It passes on AutoIt 3.3.16.1 in both x86 and x64: the x64 entry point is not installed on the author's machine, so every 64-bit number in this kit comes from an AutoIt3_x64.exe extracted next to an Include folder of the same version and run read-only. The audio suite of 0.6.1 has been measured that way more often than any other: WebView2_AV_SelfTest.au3, 40 checks, 0 failed, 92.4 s under AutoIt3.exe and 19.7 s under the extracted interpreter on browser 156.0.4314.58 - an earlier run on browser 156.0.4314.50 gave the same two numbers in the other order, 13.0 s and 93.1 s, 59.7 s of that x64 run being Chromium coming up cold. The spread is the machine, not the bitness; the properties are the same in both. It is the one file of the kit that asks the browser for the microphone and the camera - it answers every request DENY itself, and the tone it plays to test the playing state stays inaudible behind the view mute. Its navigation wait now follows the answer of its own create, because a run behind the rest of the kit lost a local file page to a 30 s clock the machine had not been paid for: a create past twelve seconds widens that one ceiling to 120 s and prints that it did. The fourth suite, WebView2_Startup_SelfTest.au3, is the create path of 0.6.2: twenty-nine headless checks over the two calls, the refusal of a view that is not ready, the borrowed environment and its reference counts - 29 passed, 0 failed on both bitnesses, and the refcount checks are what turn a leak in __WV2_EnvShareRetire into a failure. That suite used to judge a ratio between the answer of CreateStart and the length of the wait; on a warm machine the ratio called a correct implementation a regression, so the two durations are now printed and the gate is the ready flag plus $WV2_ERR_NOT_READY. The fifth suite, WebView2_Life_SelfTest.au3, is the document of 0.6.3: forty-seven checks over the order the four lifecycle events arrive in, the two BOOLs, the navigation ids, and window.close() in both directions - 47 passed, 0 failed, 0 leftovers under AutoIt3.exe and under the extracted AutoIt3_x64.exe, with the same sequences in the two logs. The sixth, WebView2_Theme_SelfTest.au3, is the appearance of 0.6.4: fifty-two checks - 52 passed, 0 failed, 57.4 s under AutoIt3.exe of which 44.8 s was the create, and 19.4 s with a 6.4 s create under the extracted interpreter. Its colour

checks read the four bytes back one channel at a time, twice, on each interpreter, which is the only way a swapped pair cannot pass, and its download check looks for the bytes of a file in the folder the profile named and for their absence in the Downloads folder of the user. The seventh, WebView2_Env_SelfTest.au3, is the environment of 0.6.5: fifty-one checks over the one object the UDF hands the runtime to be read - 51 passed, 0 failed, 126.2 s under AutoIt3.exe with a 5.2 s create and 58.6 s under the extracted interpreter with a 12.8 s one. Two of them cannot be faked by a round trip: the token written into the browser arguments has to come back out of navigator.userAgent whole, and the language has to move navigator.language away from what the machine answers on its own, which is why the file asks the machine first and gates the change. A version pinned to a build the machine has not got must fail the create and name the call; a second view started with the same options must keep the flag while the first keeps its own agent; and the view started with nothing recorded must still answer the browser's own string, because that is the branch every script of the last five releases is on. Use any of the seven self-tests as the regression gate for any change described in chapter 5, and add the new check before the implementation when you can.

WebView2_Media_Browser.au3 --auto is a third gate, and the only one that asserts on what the user sees: it drives its own window - host mute, play, pause, stop, element full screen, unmute - then reads its three labels back with GUICtrlRead, so a band that never moved fails instead of passing on stale text. 23 checks, 0 failed, 6.3 s warm under AutoIt3.exe (19.2 s cold) and 38.5 s under the extracted AutoIt3_x64.exe. Run without --auto it is the debugging view the audio surface was built for: the document band and the element band are allowed to disagree, and the line under the page says which event moved which.

References

- **WebView2 API reference**
  https://learn.microsoft.com/microsoft-edge/webview2/reference/win32/
- **ICoreWebView2Controller**
  https://learn.microsoft.com/microsoft-edge/webview2/reference/win32/icorewebview2controller
- **WebView2.h, WebView2 SDK (NuGet, MIT)**
  the authoritative vtable order
- **Microsoft x64 calling convention**
  https://learn.microsoft.com/cpp/build/x64-calling-convention
- **Forum thread: Microsoft Edge - WebView2**
  https://www.autoitscript.com/forum/topic/204362-microsoft-edge-webview2/
- **Ralle1976/Autoit-WebView2-UDF**
  https://github.com/Ralle1976/Autoit-WebView2-UDF
- **Numeric1 on the AutoIt forum**
  https://www.autoitscript.com/forum/profile/118605-numeric1/
- **PDFX, the UDF that typeset this guide**
  https://www.autoitscript.com/forum/topic/213930-udf-pdfx/

License

MIT, Copyright (c) 2026 Numeric1. Keep the LICENSE file and the Author / License lines of every source header in any copy or redistribution.

WebView2 itself is Microsoft's. The interface identifiers and vtable layout used here follow their SDK, which is MIT-licensed; no Microsoft code is reproduced in this package.

COLOPHON

This PDF is typeset with PDFX 1.4.0, the author's other UDF: contents, bookmarks, links, the cover and every figure are generated with it. Full URLs are listed in the References above and are clickable.