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 | 40 functions | 111 checks | 14 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	RUNTIME	LICENCE
0.3.0	AutoIt 3.3.16+	MIT

THE STACK

AutoIt script
GUI or --auto suite

▼

WebView2.au3
40 functions, pure AutoIt

▼

EmbeddedBrowserWebView.dll
part of Windows

▼

Edge WebView2 runtime
already installed

BY THE NUMBERS

public functions	40
self-test checks	111
handler slots	14
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 0.3.0 - October 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 03

02 Handler objects in a global heap 05

03 Vtable ranks and WebView2.h 07

04 x86 and x64: put_Bounds 09

05 Adding a method in six steps 11

06 Diagnosing a failure 13

07 Threading and the message pump 15

08 Limits, pitfalls, references 17

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: 40 functions, no DLL, no OCX, no installer. This guide is its maintenance manual.

40 PUBLIC FUNCTIONS — pure AutoIt, no loader	**111** SELF-TEST CHECKS — no network, no leftovers
14 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.

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.

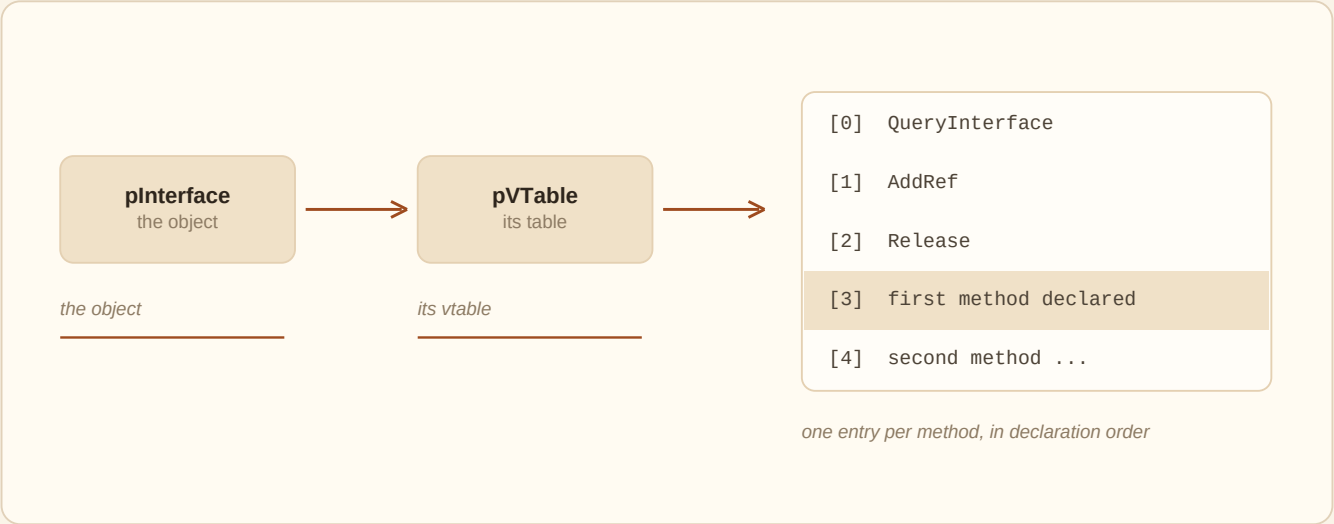


Figure 1. *Two dereferences: the object points at a table, the table points at code. Slot 3 is the first real method; slots 0 to 2 are always QueryInterface, AddRef and Release.*

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 fourteen handler objects of the slot (chapter 2).
5. **Environment** Call CreateWebViewEnvironmentWithOptionsInternal and pump until the environment callback answers, with a 15 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.

NEXT CHAPTER 02 Handler objects in a global heap

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, fourteen 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 224 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.

03

COUNTED FROM THE HEADER

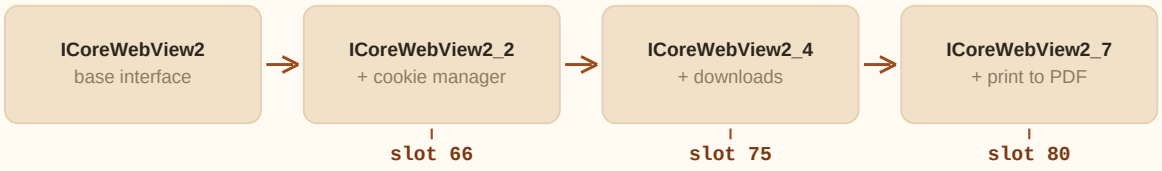
Vtable ranks and WebView2.h

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
```



Each version adds methods after the previous one: the chain is counted, never guessed.

Figure 3. *The three ranks this UDF calls raw, and the interface version that carries each one.*

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

CONSTANT	SLOT	WHAT IT IS
$__WV2_VT_QI	0	IUnknown::QueryInterface
$__WV2_VT_RELEASE	2	IUnknown::Release
$__WV2_VT_PUTBOUNDS	6	ICoreWebView2Controller::put_Bounds (x64)
$__WV2_VT_GETCOOKIE	66	ICoreWebView2_2::get_CookieManager
$__WV2_VT_ADDDL	75	ICoreWebView2_4::add_DownloadStarting
$__WV2_VT_REMOVEDL	76	ICoreWebView2_4::remove_DownloadStarting
$__WV2_VT_PRINTTOPDF	80	ICoreWebView2_7::PrintToPdf
$__WV2_VT_DLA_OP	3	DownloadStartingEventArgs::get_DownloadOperation
$__WV2_VT_DLA_CANCEL	5	DownloadStartingEventArgs::put_Cancel
$__WV2_VT_DLA_PATH	6	DownloadStartingEventArgs::get_ResultFilePath

CONSTANT	SLOT	WHAT IT IS
$__WV2_VT_DLA_PATHPUT	7	DownloadStartingEventArgs::put_ResultFilePath
$__WV2_VT_DLO_URI	9	DownloadOperation::get_Uri
$__WV2_VT_COOKIEDALL	10	ICoreWebView2CookieManager::DeleteAllCookies

Every number in that table 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.

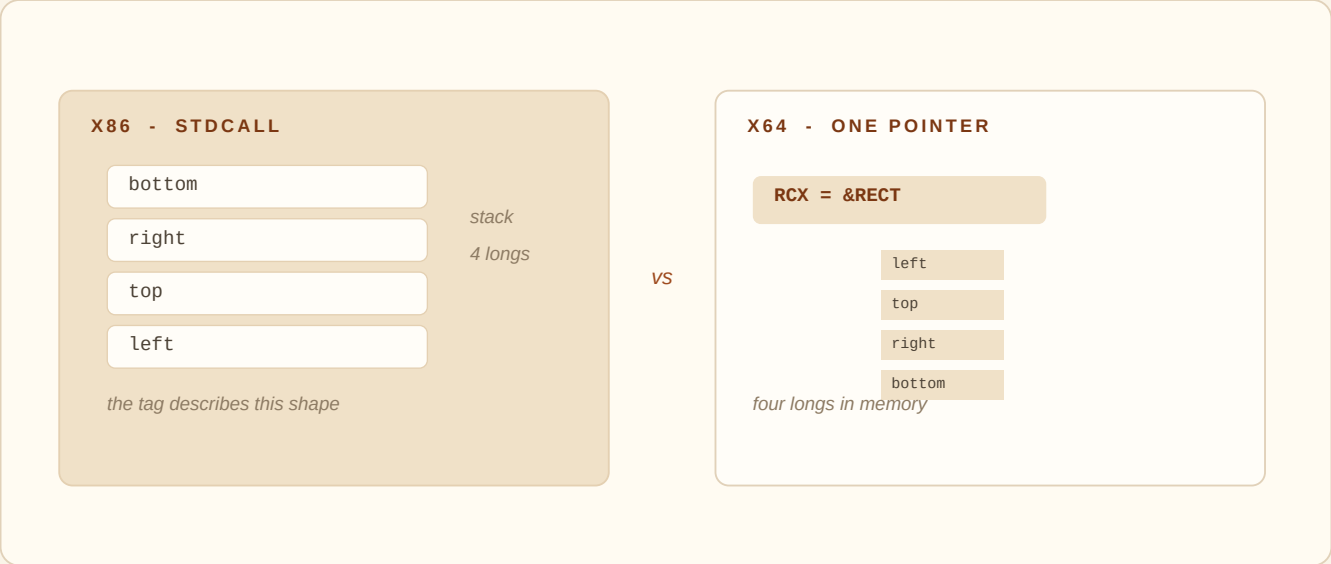


Figure 4. *The same declaration, two wire formats. Giving the callee four longs on x64 makes it dereference the first one as an address: the process terminates.*

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_VT_PUTBOUNDS)
    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.

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_VT_GETCOOKIE = 66
Global Const $__WV2_VT_COOKIEDALL = 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, $__WV2_VT_COOKIEDALL)
__WV2_ReleaseExternal($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.

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; four asynchronous methods.

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 all four:

```
__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 is a strict sequence, because an interface must outlive every pointer taken from it:

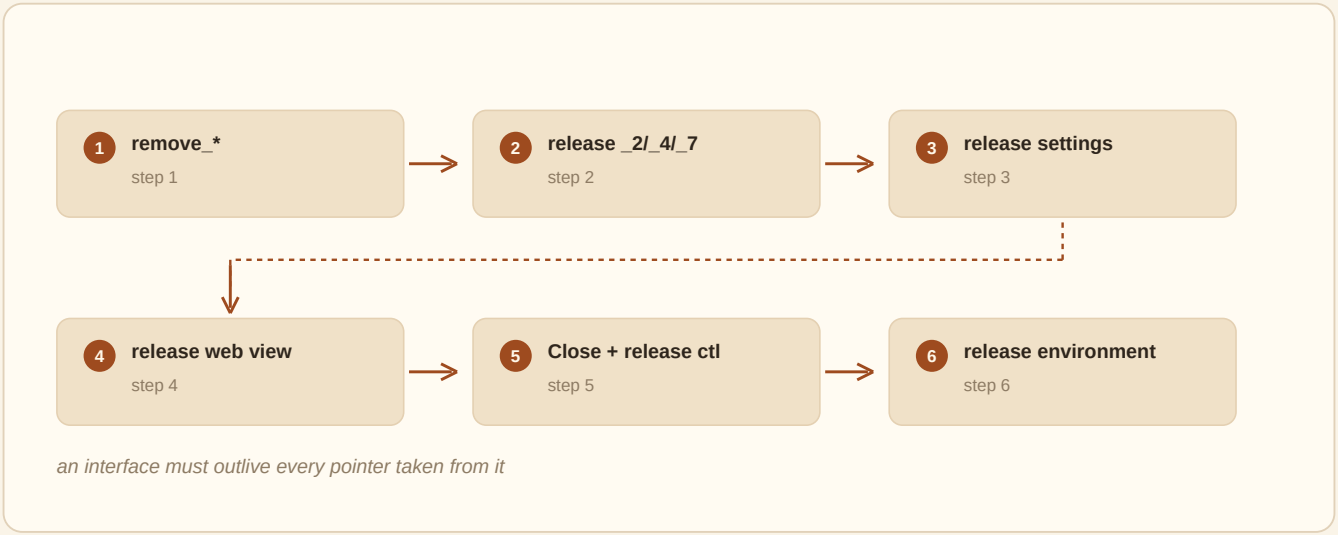


Figure 5. *The six steps of _WV2_Destroy, in order. Reversing the fourth and the second leaves the runtime holding an interface into freed memory.*

Reversing the fourth and the second leaves the runtime holding an interface into freed memory; the crash usually happens at exit, far from the cause.

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 objects per view	14	$__WV2_SLOT_COUNT
Queued events per view	64	$__WV2_QCAP, oldest dropped
Create timeout	15 s	$WV2_CREATE_TIMEOUT
Navigation timeout	30 s	$WV2_NAV_TIMEOUT
Script / 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.
- **Cookies only** _WV2_ClearCookies clears cookies. Cache, history and localStorage need ICoreWebView2Profile2::ClearBrowsingData, which the view does not expose.
- **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.
- **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.
- **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.

Testing

WebView2_Example_GUI.au3 /auto runs 111 checks over the whole API without network access (file:// and NavigateToString pages) and leaves nothing on disk. It passed on AutoIt 3.3.16.1 in x86 and x64. Use it as the regression gate for any change described in chapter 5, and add the new check before the implementation when you can.

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.