AUTOIT UDF BY NUMERIC1 - QUICK START

WV2Bridge

3.1.1

Authenticated, promise-based RPC between a WebView2 page and AutoIt. The page writes await au3.json('sum', 40, 2); AutoIt answers from its own loop. No switch on a message string, no hand-made reply convention, no open door.

AutoIt 3.3.16+ | No DLL | No installer | WV2B/3 wire | 39 functions

```
#include "WV2Bridge.au3"

_WV2B_Init($iWV, @ScriptDir & "\web", "app.local")
_WV2B_Register("sum", "_H_Sum", 0, 1)

While GUIGetMsg() <> $GUI_EVENT_CLOSE
    _WV2B_Poll()
    Sleep(10)
WEnd
_WV2B_DestroyAll()

; and in the page:
const n = await au3.json("sum", 40, 2);   // 42
```

Two lines of setup, one line in the loop. That is the whole bridge.

WHAT IS ON THE OTHER SIDE

VERSION	PROTOCOL	LICENCE	AUTHOR
3.1.1	WV2B/3	MIT	Numeric1

WHAT YOU NEED

Windows 10 or 11
the WebView2 runtime is already there

AutoIt 3.3.16+
x86, measured on this machine

WebView2.au3 0.6.2
bundled in this kit

BY THE NUMBERS

AutoIt functions	39
page API members	19
self-test checks	86
one call at a time	33.5 ms
48 calls pipelined	1.89 ms

Every number here was measured on one machine (Windows 10 19045, AutoIt 3.3.16.1 x86, WebView2 Runtime 156.0.4314.58). The checks run from the kit itself - 86 in WV2Bridge_SelfTest.au3, 11 through a real page; the timings come from the audit harness on the same build. Nothing is estimated.

THE SIX CHAPTERS

01 **One call, end to end**
the wire, then a running program

02 **The page side**
call, json, send, on, b64, the error object

03 **The AutoIt side**
register, arguments, return, Poll, teardown

04 **Bytes, files, slow work**
strict base64, upload, defer and resolve

05 **What the bridge refuses**
the secret, the origin, the forged frame

06 **Flow control and failures**
the ring of 64, latency, what breaks

https://www.autoitscript.com/forum/

HOW TO USE THIS BOOKLET

Start here

Six chapters. The first one is already a working program; the next five add one idea at a time.

01 One call, end to end 03

02 The page side 06

03 The AutoIt side 08

04 Bytes, files and slow work 10

05 What the bridge refuses 12

06 Flow control, latency, failures 14

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

BEFORE YOU TYPE

01 Windows 10 or 11 with the WebView2 runtime - Windows 11 ships it, Windows 10 gets it with the Edge update.

02 AutoIt 3.3.16 or later - every program here ran under AutoIt3.exe (x86).

03 WebView2.au3 and WV2Bridge.au3 next to your script - both are in this kit, nothing to register.

THE SHAPE OF IT

One library, two example programs, and the WebView2 UDF it rides on - bundled, so the kit runs as it is. The code between the chapters is cut from the files in the kit by line number, so what you read is what you run. 3.1.0 is the audited edition: eight defects were found by measurement and closed, and chapters 5 and 6 quote the number each fix moved.

39
PUBLIC FUNCTIONS
in the library

86
SELF-TEST CHECKS
headless, exit code

0
DLLS OR INSTALLERS
pure AutoIt

128
BITS OF SECRET
one per view, random

READING PATHS

Write the first call	01 - 02 - 03
Ship an upload or a long job	04
Prove it holds under load	05 - 06

```
AutoIt3.exe WV2Bridge_SelfTest.au3  ->  86/86 passed
```

01

THE WIRE AND THE FIRST PROGRAM

One call, end to end

A page asks for something, AutoIt answers, and both know who they are talking to. This chapter shows the whole mechanism before a single API name appears.

What the bridge replaces

The WebView2 UDF gives you a queue of web messages: you get a string and it is your problem to know what it meant. Every project that grows out of that ends up with the same three things - a Switch on the message, a reply convention only you understand, and no answer to the page that is not yours. WV2Bridge replaces all three with named functions and promises.

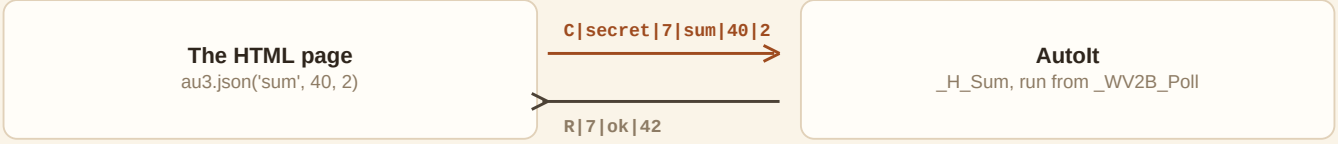
APPROACH	WHAT YOU WRITE
Without the bridge	Every call is a string you invent, a prefix you parse, a reply id you hand-write, and a page that can post anything to chrome.webview and be answered.
With the bridge	The page writes await au3.json('sum', 40, 2). AutoIt registers _H_Sum once, and the library carries the secret, the id, the error code and the timeout for you.

The wire, in one page

Everything travels as one string per message, fields separated by Chr(31) - a character no HTML page produces by accident. Six frame types, and that is the whole protocol.

FRAME	WHAT IT IS
C|secret|id|name|arg...	a call: answer it with an R frame (page -> AutoIt)
N|secret|0|name|arg...	a one-way send: no answer, no slot held (page -> AutoIt)
H|secret|protocol|href	the page loaded the bridge (page -> AutoIt)
R|id|ok|payload	the answer, already encoded as text or JSON
R|id|err|code|message	the refusal, with the code your handler returned
E|name|payload A|count	an event you pushed, and the acknowledgement of n one-way frames

The diagram, in one page



Nothing in the middle is registered, polled over a network, or written by hand. The id comes from the page, the secret comes from AutoIt and is injected before your page code runs, and the answer is matched back to the promise that asked for it.

The first program

This is 01_minimal_rpc.au3 from the kit, the setup part only. It maps a folder as https://app.local/, injects the bridge into it, registers two functions and navigates.

```
    Local $sWeb = @ScriptDir & "\web_min"
    DirCreate($sWeb)
    If Not FileExists($sWeb & "\index.html") Then FileWrite($sWeb & "\index.html", _Page())

    If $g_bCheck Then
        $g_hGui = GUICreate("WV2Bridge - minimal", 620, 400, -2000, -2000)
    Else
        $g_hGui = GUICreate("WV2Bridge - minimal", 620, 400)
    EndIf
    GUISetState(@SW_SHOW, $g_hGui)
```

Lines 36-45 of 01_minimal_rpc.au3: the folder that will be mapped, and the window the page fills.

```
    ; Map the folder as https://app.local/, inject the bridge, and lock the view to that origin.
    If Not _WV2B_Init($g_iWV, $sWeb, "app.local") Then
        If $g_bCheck Then
            ConsoleWrite("_WV2B_Init failed, @error = " & @error & "  " & _WV2_GetLastErrorMessage() & @CRLF)
            Exit 2
        EndIf
        MsgBox(16, "Bridge", "_WV2B_Init failed, @error = " & @error & @CRLF & _WV2_GetLastErrorMessage())
        Exit 1
    EndIf

    ; (name, AutoIt function, view 0 = all views, min args, max args)
    ; The arity and the argument types are checked before your code runs.
    _WV2B_Register("sum", "_H_Sum", 0, 1)
    _WV2B_Register("save", "_H_Save", 0, 2, 2)
    _WV2B_SetHelloHandler("_OnHello")   ; the page has loaded the bridge

    ; Repeating, in both modes: in check mode the page may register its listener after the first
    ; tick, and a one-shot would then be missed - a wait must not depend on that race.
    _WV2B_AddTask("_Tick", 1000)   ; AutoIt -> page, with no request from the page

    _WV2_Navigate($g_iWV, "https://app.local/index.html")
```

Lines 57-77: _WV2B_Init maps and injects and locks the origin, then two handlers, one pushed task, and the navigation. The error branches are left out here - in /check they print why the run aborted.

And the handler

```
Func _H_Sum($aArgs)
    Local $fTotal = 0, $v
    For $i = 1 To $aArgs[0]
        $v = _WV2B_ArgNum($aArgs, $i)
        If @error Then Return _WV2B_Fail("argument " & $i & " is not a number", 422)
        $fTotal += $v
    Next
    Return $fTotal   ; the page receives it as a string; au3.json() parses it
EndFunc   ;==>_H_Sum
```

$aArgs[0] is the count, $aArgs[1..n] are the arguments as text. _WV2B_ArgNum refuses a non-number before you have to look at it.

RUN IT

Double-click 01_minimal_rpc.au3, or run AutoIt3.exe 01_minimal_rpc.au3. Four buttons: a call, a call the handler refuses, an upload of four bytes, and a path traversal that must be refused. A line appears every second, pushed by AutoIt - the page never asked for it.

Run it as a test

The same program has a second mode. It drives the page instead of a mouse, and the exit code is the number of failed checks - 0 means the kit works on this machine, so it drops into CI as it is.

```
AutoIt3.exe 01_minimal_rpc.au3 /check    ->    exit 0
```

Eleven assertions: the round trip, extra arguments, the handler's own 422, an unknown name, refused arity, a refused traversal, the one-way send, the pushed event, the four bytes on disk, nothing written beside the folder, and zero dropped frames.

What to change for your own app

1. **Register** one _WV2B_Register per function the page may call; the name the page uses is the first argument.
2. **Write** a handler that reads its arguments with the _WV2B_Arg* helpers and returns a string, a number, an array, a Map or Binary - the library encodes it.
3. **Poll** _WV2B_Poll() inside your message loop. Handlers run there, never inside a runtime callback.
4. **Tear down** _WV2B_DestroyAll() when the window closes, in the loop, not in an OnAutoItExit handler.

NEXT CHAPTER 02 The page side

02

EVERYTHING window.au3 GIVES YOU

The page side

The bridge script is injected before your own code runs, so window.au3 exists in every page you serve from the mapped folder. There is nothing to include in the HTML.

The nineteen members

MEMBER	WHAT IT DOES
au3.call(name, ...args)	Promise of the answer as text
au3.json(name, ...args)	the same answer parsed as JSON
au3.send(name, ...args)	one way, no promise; true if the frame left, false if the queue is full
au3.on(name, fn)	listen for an event; returns an unsubscribe function
au3.onJson(name, fn)	the same, payload parsed
au3.once(name, fn)	one event, then unsubscribe
au3.off(name, fn)	remove a listener you kept
au3.with({timeout, maxInflight, ...})	a second client with different limits
au3.ready	Promise of true once AutoIt answered a ping
au3.version	the version string of the injected script
au3.hosted	true when chrome.webview is present
au3.b64.encode / decode	text <-> base64, for the upload-style handlers
au3.timeout	get or set the call timeout in ms, default 30 000 (0 = none)
au3.maxInflight	get or set the call window, clamped to 48
au3.sendWindow	get or set the one-way window, clamped to 48
au3.sendPending	frames still waiting for the host to acknowledge
au3.sendQueue	get or set the bounded queue behind them
au3.maxInflightCeiling	48 - the clamp, so a UI can show it
au3.hostAcks()	did the host ever send an A frame (false = older host)

Arguments: what the page may hand over

- **Plain values** numbers, strings, booleans, null, objects and arrays are encoded for you. An object or array arrives in AutoIt as JSON text; decode it with _WV2B_ArgJson or _WV2B_JsonDecode.
- **Binary** a Uint8Array, ArrayBuffer or Blob arrives as base64, and the handler reads it with _WV2B_ArgBinary.
- **Two characters you must not send** arguments are separated by Chr(31), so it is stripped from every field, and a string holding Chr(0) is refused with code 400 - PostWebMessageAsString truncates at the NUL, measured. Send base64 for anything binary.

- **Arity is checked first** _WV2B_Register('sum', '_H_Sum', 0, 1) means at least one argument, and a page that sends none gets code 400 without your handler being called.

The error object

A rejection is always an Au3Error with a numeric code and a message. The code is yours when the handler sets it; the library's when the refusal happens before the handler.

CODE	NAME	WHEN THE PAGE SEES IT
-1	not hosted	the page is not inside WebView2, so there is no host to answer
-2	TimeoutError	no answer within au3.timeout; e.name is set so you can branch on it
400	bad request	message over the size limit, more than 64 arguments, wrong arity, a NUL in a payload
403	refused	the view is not on an allowed origin - the host also returns it from _WV2B_Emit
404	not found	no handler registered under that name for this view
500	internal	the AutoIt function behind the name does not exist, or a reply held a NUL
1	no code given	a handler that returned _WV2B_Fail('text') without a code
any	yours	whatever your handler passes to _WV2B_Fail - 422 is the one the examples use for a bad argument

```
try { say("sum = " + await au3.json("sum", 40, 2)); }
catch (e) { say("refused: code " + e.code + " " + e.message); }
```

From the page in 01_minimal_rpc.au3. e.code is a number, so a UI can branch on it.

Events pushed by AutoIt

_WV2B_Emit($iWV, 'tick', $sPayload) sends to one view, _WV2B_Broadcast('tick', $sPayload) to every attached view. au3.on and au3.onJson receive it; the unsubscribe function they return is what keeps a component's teardown honest.

```
au3.onJson("tick", m => {
  live.textContent = "AutoIt pushed: " + m.calls + " calls served, " + m.avgMs + " ms on the average";
});
```

From the same page. The example re-emits every second, so a listener registered late still sees one.

> **ONE-WAY, WHEN YOU WANT IT**
> au3.send is for state the page pushes and never needs an answer for - scroll position, telemetry, 'the panel opened'. It costs one leg instead of two and holds no slot, so it cannot time out. It is still flow controlled: chapter 6.

NEXT CHAPTER 03 The AutoIt side

03

THE FORTY FUNCTIONS, GROUPED

The AutoIt side

Attach a view, register handlers, run the loop, tear down. Everything else is a variation on those four.

Attach and configure

CALL	WHAT IT DOES
_WV2B_Init($iWV, $sFolder, $sHost)	map a folder as https://$sHost/ (default app.local), inject the script, lock the view to that origin
_WV2B_Attach($iWV, $sOrigins)	the same for a view that serves itself - NavigateToString, a real site; $sOrigins defaults to *
_WV2B_Detach($iWV)	let go of one view: script, mapping, handlers, pending calls
_WV2B_Destroy($iWV) / _WV2B_DestroyAll()	detach and clean the tables - call it before GUIDelete
_WV2B_SetHelloHandler($sFunc)	your Func Hello($iWV, $sHref, $sProtocol): the page loaded the bridge
_WV2B_SetEventSink($sFunc)	everything that is not a bridge message, handed to you instead of dropped

Register and run

CALL	WHAT IT DOES
_WV2B_Register($sName, $sFunc, $iWV, $iMin, $iMax)	one name, one handler; $iWV 0 means every view, $iMax -1 means no upper bound
_WV2B_Unregister($sName, $iWV) / _WV2B_IsRegistered	remove a name, or ask whether a view answers to it
_WV2B_Poll([$iWV])	drain the queue and run the handlers - this is the whole runtime
_WV2B_AddTask($sFunc, $iMs, $bOnce)	a timer that fires inside Poll, so it cannot race your handlers
_WV2B_Emit($iWV, $sName, $vPayload)	push an event to one view; _WV2B_Broadcast does all of them
_WV2B_CurrentView() / _WV2B_CurrentName()	inside a handler: which view asked, and under which name

What a handler may return

- **A string** and the page gets exactly that; au3.call resolves with it.
- **A number, a boolean, an array or a Map** and the library encodes it as JSON text, so au3.json parses it back.
- **Binary** and it travels as base64, ready for au3.b64.decode or a Uint8Array.
- **_WV2B_Fail($sMessage, $iCode)** and the page's promise rejects with that code and message.
- **_WV2B_Defer()** inside the handler: it returns a ticket string, your return value is ignored, and you answer later with _WV2B_Resolve($sTicket, $vValue) or _WV2B_Reject($sTicket, $sMessage, $iCode).

```
Func _H_Save($aArgs)
    Local $sName = _WV2B_Arg($aArgs, 1)
    ; A file name that walks out of the folder is refused, not sanitised.
    If StringInStr($sName, "\") Or StringInStr($sName, "/") Or StringInStr($sName, ":") Or $sName = "" Then _
            Return _WV2B_Fail("file name must be a plain name", 400)
    Local $b = _WV2B_ArgBinary($aArgs, 2)
    If @error Then Return _WV2B_Fail("second argument is not base64", 400)
    Local $hF = FileOpen(@ScriptDir & "\web_min\" & $sName, 2 + 16)   ; overwrite + binary
    If $hF = -1 Then Return _WV2B_Fail("cannot write the file", 500)
    FileWrite($hF, $b)
    FileClose($hF)
    Return '{"saved":"' & $sName & '","bytes":' & BinaryLen($b) & '}'
EndFunc   ;==>_H_Save
```

_H_Save from the kit: a plain-name check, a strict base64 read, a binary write. Three refusals, each with the code the page will see.

Reading arguments

HELPER	WHAT YOU GET
_WV2B_Arg($aArgs, $i, $vDefault)	the raw text; $vDefault when the index is past the count
_WV2B_ArgNum($aArgs, $i, $vDefault)	a number, with Inf and NaN refused; @error 1 missing, 2 not a number
_WV2B_ArgJson($aArgs, $i)	a Map or 1D array decoded from JSON; @error 2 when it is not valid JSON
_WV2B_ArgBinary($aArgs, $i)	base64 decoded to Binary - chapter 4 explains why this one is strict

Diagnosis

CALL	WHAT IT TELLS YOU
_WV2B_GetStats()	a Map: calls, errors, refused, forged, blocked, dropped, deferred, pending, views, slow, avgMs, maxMs, tasks, version
_WV2B_SetLogger($sFunc, $iLevel)	every call, refusal and drop through your Func Log($iLevel, $sMsg)
_WV2B_CallCount() / _WV2B_PendingCount()	two numbers, no Map to unpack
_WV2B_GetJsSource() / _WV2B_WriteJs($sFile)	the injected script itself, for MODE_FILE or for an audit
_WV2B_SelfTest()	the 86 checks; they live in WV2Bridge_SelfTest.au3 beside the library, which runs nothing itself

TEARDOWN, AND WHY IT IS IN THE LOOP

Call _WV2B_DestroyAll() after the GUI loop ends and before GUIDelete. Not in an OnAutoItExit function: releasing a WebView2 object needs the message loop alive, and at exit it is not. A view destroyed behind the library's back is detected on the next Poll and detached, so a recycled handle cannot inherit another view's handlers.

NEXT CHAPTER 04 Bytes, files and slow work

04

BYTES, FILES AND WORK THAT TAKES TIME

Bytes, files and slow work

Two things break a naive bridge: binary payloads and answers that are not ready when the handler returns. Both have a first-class path.

Binary in, binary out

A Uint8Array, ArrayBuffer or Blob on the page becomes base64 on the wire. On the AutoIt side _WV2B_ArgBinary decodes it - and this is where a shortcut cost a whole audit cycle, because crypt32's CryptStringToBinaryW invents bytes for input it should refuse.

@EXTENDED	REASON	WHAT IT MEANS IN YOUR HANDLER
1	outside the alphabet	a character that is not A-Za-z0-9+/= - 'a!b=' is refused, not repaired
2	impossible length	4n+1 characters can never be base64; crypt32 decoded a lone 'A' to 0x04
3	padding off a boundary	'SGl0QQ=' has its = in the wrong place
4	crypt32 still refused it	well formed to us, rejected by the system

@error 1 means the argument is missing, @error 2 means it is not valid base64, and the reason is in @extended - so a handler can put that reason in the refusal the page sees instead of a bare 'bad input'.

```
Local $b = _WV2B_ArgBinary($aArgs, 2)
If @error Then Return _WV2B_Fail("base64 rejected (reason " & _Reason(@extended) & ")", 400)
```

The pattern from the audit driver: propagate the reason, keep the code the page branches on.

Files: write inside the folder, refuse the rest

The kit's _H_Save takes a plain name and a byte array. The name is checked before anything touches the disk: a backslash, a slash, a colon or an empty name is refused with 400. That is a refusal, not a sanitise - the library never rewrites what the page asked for.

- **Measured** '../evil.txt' never reaches the file system, and the check that no such file exists next to the script is one of the eleven assertions in /check.
- **Yours to decide** the bridge does not sandbox your handlers. If a name may be arbitrary, resolve it and then verify it is still under the folder you meant.
- **Blobs back out** return Binary from a handler and the page gets base64; au3.b64.decode turns it into a Uint8Array again.

Answers that are not ready yet

Handlers are synchronous: a slow handler delays every other call in the same Poll. For work that takes seconds - a PDF, a query, a download - take the ticket and answer later.

```
Global $g_sTicket = ""

Func _H_Build($aArgs)
    $g_sTicket = _WV2B_Defer()      ; the page keeps waiting, your return value is ignored
    _WV2B_AddTask("_Finish", 2000, True)   ; or any later event
    Return ""
EndFunc

Func _Finish($iTask)
    _WV2B_Resolve($g_sTicket, """done""")
    ; or: _WV2B_Reject($g_sTicket, "the build failed", 500)
EndFunc
```

The page's own timeout still applies, and a deferred call keeps its slot in au3.maxInflight until it is answered. A ticket nobody resolves expires after 5 minutes.

Time inside the loop

- **_WV2B_SetLimits($iMaxChars, $iMaxArgs, $iBudgetMs, $iMaxPerPoll, $iSlowMs, $iDeferTtlMs)** the six knobs. Defaults: 4 194 304 characters per page message, 64 arguments, 40 ms per view per Poll, unlimited calls per Poll, 250 ms before a handler is logged as slow, 300 000 ms of deferred TTL.
- **Budget, not a thread** Poll stops taking new calls when the view's budget is spent and comes back on the next iteration, so a slow handler cannot starve the window's other messages.
- **Tasks** _WV2B_AddTask fires inside Poll. It cannot interleave with a handler, which is the reason it exists instead of AdlibRegister.

8 MB, MEASURED

A reply of 8 MB crosses the bridge whole in 1166 ms. There is no outbound size ceiling to design around and no chunking protocol to write; the inbound limit is the one that protects you from a page.

NEXT CHAPTER 05 What the bridge refuses

05

THE MODEL, AND WHAT WAS MEASURED AGAINST IT

What the bridge refuses

The bridge is not a message queue with names. It is an authenticated channel, and 3.1.0 is the edition where that claim was tested rather than asserted.

The secret

Each attached view gets 128 random bits from RtlGenRandom, hex-encoded, carried by the page script that _WV2_AddScriptToExecuteOnDocumentCreated injects before any of your page code runs. Every frame the page sends carries it, and a frame without the right one is counted as forged and never answered.

- **Bound, not looked up** the script captures wv.postMessage.bind(wv) at injection time. A page that hooks postMessage afterwards sees nothing leave: measured 0 frames observed, with the hook installed before the bridge ran.
- **Top frame only** the script defines window.au3 only when the frame is the top one and the origin is allowed. A cross-origin iframe can see chrome.webview but its posts never reach the host.
- **Answers are conditional** replies and pushed events are posted only while _WV2_GetSource still reports an allowed origin, so a page that navigated away cannot collect what was meant for it.

The origin

The host-side check costs 0.115 ms per call, which is why there is no cache of the current URL and nothing to invalidate. Scheme and host are compared case-insensitively, the path case-sensitively, the way browsers compare them.

What the attack run found

The table below is the output of the audit harness that drove this build - the same seven attempts, run against the library in this kit on the machine named on the cover. It is not shipped with the kit; the two lines of proof that ship are the 86 self-test checks in their own file and the 11 page assertions of chapter 1, and they cover the guards that matter.

WHAT WAS TRIED	WHAT HAPPENED	WHY
A page hook installed before the bridge	0 frames observed	postMessage captured and bound at injection
A frame with the wrong secret	counted as forged, never answered	2 in the run, both silent to the page
A call from a foreign iframe	no window.au3, no reply	top-frame and origin guards in the script
sum('x')	code 422 from the handler	arity and type checked before your code runs
save('../evil.txt', ...)	code 400, no file created	the handler refuses, the check proves the absence
A payload containing Chr(0)	refused on the page, error frame on the host	PostWebMessageAsString truncates - measured
200 calls in flight	clamped to 48, 200 answered, 0 timeouts	the ring of 64 can no longer be overrun

What it does not protect against

> **YOUR OWN ORIGIN**
> A script running inside a page you trust can call your handlers - that is XSS and no bridge fixes it. Use a CSP, escape what you render, validate every argument in the handler. The bridge's job is to make sure whoever calls is a view you attached, on an origin you allowed.

- **Not a web server** there is no port, so there is nothing for a LAN attacker to reach. Everything is in-process, through the runtime's own message channel.
- **Not an authorisation layer** the bridge knows which view asked, not which user. If two of your pages may do different things, register different handlers per view.
- **Not a substitute for checking paths, SQL or shell arguments** the same rules as anywhere else in AutoIt.

Built-ins, and the reserved prefix

au3.ping, au3.list and au3.info are written straight into the registry by the library, so they exist before you register anything - and _WV2B_Register refuses a user name starting with 'au3.', so no handler can shadow them. au3.list answers with the names the current view can call, au3.info with the stats Map and the protocol version. Both are worth calling the moment a page misbehaves.

NEXT CHAPTER 06 Flow control, latency, failures

06

FLOW CONTROL, LATENCY, AND WHAT BREAKS

Flow control, latency, failures

The bridge sits on a queue that drops messages. This chapter is the measurement of that queue, and the list of what actually goes wrong in practice.

The ring of 64

WebView2.au3 keeps 64 pending web-message events per view and drops the oldest silently when it overflows. Measured: 200 calls in flight produced exactly 64 answers. The bridge therefore clamps both windows to 48 and acknowledges one-way frames, so a page can never ask for a number that guarantees losses.

DIRECTION	HOW IT IS BOUNDED	WHAT THAT BUYS YOU
Calls	au3.maxInflight, clamped to 48	extra calls wait in the page's own queue, they are not lost
One-way sends	au3.sendWindow plus a bounded au3.sendQueue	au3.send returns false instead of overflowing
The acknowledgement	an A|count frame, counted per view	one page's A frame can never open another page's window
Drops	stats['dropped'], from the UDF's own counter	the one number to watch in a load test

The measured floor

WHAT	MEASURED	WHAT IT DECIDES
one call at a time	33.5 ms median	the transport costs about 15 ms each way; polling is not the bottleneck
48 calls pipelined	1.89 ms per call	what a page should do for a list, not a loop of awaits
one posted frame	1.5 ms	the host side of a send
the origin check	0.115 ms	why there is no URL cache to invalidate
a reply of 8 MB	1166 ms	arrives whole - no outbound ceiling
the view, cold	2989 ms	Chromium's start-up, not your handler

The consequence for an interface: the state of the UI lives in the page. AutoIt does the authoritative work and pushes the result with _WV2B_Emit. A page that asks for everything on every paint will feel 33.5 ms per question, and no version of this bridge fixes that - it is one message in each direction through the runtime.

What actually goes wrong

SYMPTOM	CAUSE, IN PRACTICE
Nothing happens at all	the WebView2 runtime is missing, or _WV2B_Init ran before the view existed
The page has no au3	the page is not on the allowed origin, or it is in an iframe

SYMPTOM	CAUSE, IN PRACTICE
Every call rejects with 404	the name is registered for another view, or the case differs
Calls hang, then -2	you are not calling _WV2B_Poll often enough, or a handler never returned
$WV2_ERR_BUSY inside a handler	you called a WebView2 function from a runtime callback - keep the work in Poll
Answers stop after a reload	deferred tickets of the previous page are dropped on purpose; re-issue from the hello handler
A stripped build cannot find handlers	Au3Stripper: list them in #Au3Stripper_Ignore_Funcs, they are called by name
The window dies at exit	teardown ran in OnAutoItExit; move _WV2B_DestroyAll() into the loop path

How to keep it proven

1. **Au3Check** every file you ship: -q -d -w 1 -w 2 -w 7, one file per invocation.
2. **AutoIt3.exe WV2Bridge_SelfTest.au3** 86 checks, no window, exit code = failures.
3. **AutoIt3.exe 01_minimal_rpc.au3 /check** 11 assertions through a real page, exit code = failures.
4. **Watch the stats** calls, errors, refused, forged, dropped. In a load test, dropped != 0 is the only number that means the bridge lost something.

> **THE VERSION CONTRACT**
> The page script and the library are one build: the injected script returns early if window.au3.version already equals the version being injected, and an au3bridge.js left by an older release is replaced by an empty stub so it cannot overwrite the new one.

Where to read further

README.md in the kit carries the API at a glance, the design notes and the known limits. The header of WV2Bridge.au3 documents the wire format, what a handler may return and every signature - the same text the self test was written against. CHANGELOG.md lists each 3.1.0 fix with the measurement that found it.