

APL64: Using HTLC and HTLS

Contents

Overview	5
Communication between APL64 instance	5
Typical Uses	5
Client and Server Configuration	5
Local and Remote Servers.....	5
Multiple Clients and Servers.....	6
Development and Runtime Support	6
Client Actions using  HTLC.....	7
Server-side Debugging Supported	7
Simple Example	7
Local Exclusive or Public Server	8
Synchronous  HTLC actions from the same Client	9
Asynchronous  HTLC CallAsync action	9
Requests and Results associated with one Client	9
Multiple Clients accessing the same Server.....	9
 HTLC Actions	10
Call	10
CallAsync	12
Exec	18
GetClientId	20
GetServerLog	21
GetSysVar	21
GetVar	21
Help (?).....	22
ListenLocal	23

LoadWs	31
ServerOff	32
SetSysVar	32
SetVar.....	33
Stop	34
SysCall	34
SysCommand	35
Timeout	36
Variable	36
Visible	37
☐ HTLS Actions.....	38
ClearLog	38
GetLog.....	38
Help (?)	41
Listen	42
LogFileOptions	46
Pause	48
Resume	48
ServerOff	49
Stop	50
UrlInfo	50
☐ HTLCURL	51
Development and Runtime Use of ☐ HTLC and ☐ HTLS	52
Development Environment	52
Runtime Environment	52
Example #1: Using ☐ HTLS in a WRE	52
Create the WRE (C:\HtIsWreTest\Source\HtIsWreTest.ws64) workspace	52
Complete the APL64 WRE creation utility dialog.....	54
Save the WRE Creation Utility Input.....	54

Use an APL64 Instance to access the WRE-based □HTLS Server	55
Example #2: □HTLS in a CPC	56
Create the C:\HtIsCpcTest\Source\HtIsCpcTest.ws64 workspace	56
Complete the APL64 CPC creation dialog	58
Click the CPC creation dialog 'Create' button	58
Save the CPC Creation Dialog Input	58
Use the APL64 CPC Nuget package in a .Net console project	59
Run the console application using the APL64 CPC to Start the □HTLS Server	63
Use an APL64 Developer Instance to Access the □HTLS Server	63
Example #3: □HTLC in a CPC	64
Create the disk folders for the example	64
Create source workspace and public functions	64
CallDyadic	64
CallMonadic	65
CallNiladic	66
Exec	67
GetClientId	68
GetServerLog	68
GetSysVar	69
GetTimeout	70
GetUrl	70
GetVar	71
GetVisible	72
Help	73
LoadWs	73
SetSysVar	74
SetTimeOut	75
SetUrl	75
SetVar	76

SetVisible	77
SysCallDyadic.....	77
SysCallMonadic	78
SysCallNiladic	79
SysCommand	80
Complete the APL64 CPC creation dialog.....	81
Click the CPC creation dialog ‘Create’ button	82
Save the CPC Creation Dialog Info	82
Create a .Net Project to use the CPC Nuget Package	83
Install the CPC Nuget Package into the .Net Project	84
Enter the C# source code.....	85
Start an APL64 instance and use ☐ HTLS to create an HTLS server	89
Run the console application	89
Comparing ☐ HTLC/HTLS and ☐ WSE.....	89
Analogous Client and Server Environments	89
Creating the Server	90
☐ HTLC and ☐ WSE have Analogous Actions	91
Closing the Server	91
Multiple Instances	92
Performance	93
Debugging.....	93
Configuring an ☐ HTLS server to listen on an HTTPS port.....	93
Port Forwarding (For access by external clients).....	93
Firewall Configuration	94
Administrator Privileges	94
Certificate Trust and Validation.....	94
Url Binding Permissions (http.sys on Windows).....	94
Dns and Hostname Resolution	94
Router Settings (For external Https access)	95

Network Topology	95
Testing and Verification	95
□ HTLS validation and exception handling	95
Security Best Practices	96
Certificates	96
Network	96
Authentication	96

Overview

Communication between APL64 instance

The □ HTLC and □ HTLS system functions support communication between APL64 instances. Communication uses □ HTLC actions, such as Exec, Call, SysCall, CallAsync, etc.

Typical Uses

A Client APL instance creates one or more local APL instances and uses them as APL64 workspace engines with separate workspaces, functions, variables as necessary.

One or more client APL instances access a shared APL server instance, with each client updating the state of the APL server instance.

One or more client APL instances access one server APL instance which is the front-facing server which sends specific requests to additional server APL instances.

Client and Server Configuration

Instances of the APL64 Developer version can be configured as client and server using the □ HTLC and □ HTLS system functions. In this configuration the client and server instances of APL64 communicate using the [http protocol](#). The client uses a .Net [HttpClient object](#). The server uses a [.Net Kestrel server object](#) listening on a programmer-specified [url](#).

Local and Remote Servers

An APL instance as a client using the □ HTLC ‘ListenLocal’ action can create one or more local servers (on the same machine) for exclusive use by that client.

An APL instance can become a server using  HTLS 'Listen'. This server can be local or remote with respect to potential clients.

Each server includes an independent APL64 interpreter (workspace engine). A server can exist on any compatible machine connected to the Internet or local area network supporting the http/https protocol. All required server components are included in APL64.

Example: Local Exclusive Server for a Client

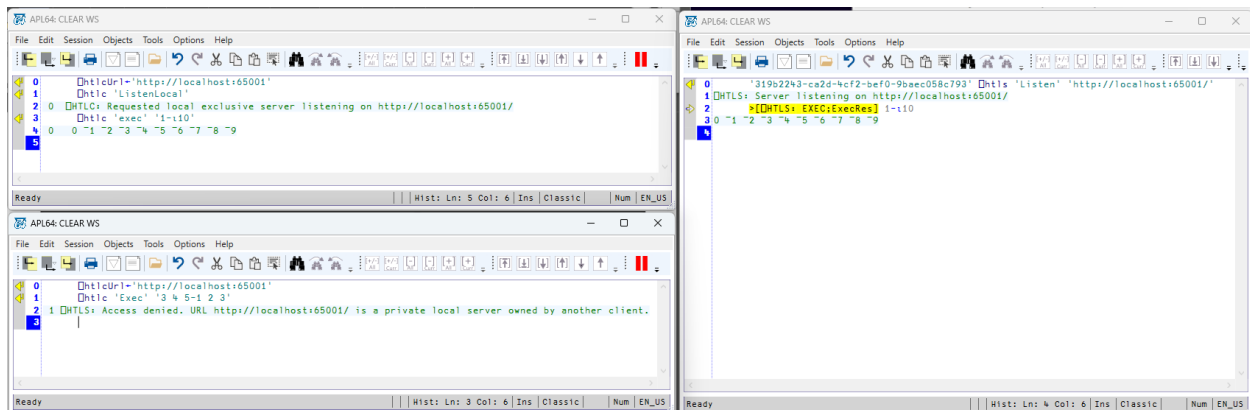
Client #1 Creates a local exclusive server and uses it:

```
 htclUrl←'http://localhost:65001'  
 htcl 'ListenLocal'  
 htcl 'exec' '1-10'
```

Client #2 attempts to access client #1 local exclusive server:

```
 htclUrl←'http://localhost:65001'  
 htcl 'Exec' '3 4 5-1 2 3'
```

Client #2 is denied access to client #1 server:



Multiple Clients and Servers

Multiple clients and servers can interact using  HTLC. A server can listen on one or more urls. A client can access more than one server.

Development and Runtime Support

 HTLC and  HTLS system functions can be used in APL64 developer version instances and in the APL64 runtime environment, including a Windows Runtime Executable (WRE) and a cross-platform component (CPC).

Client Actions using □HTLC

□HTLC actions create client requests for the server to perform server-side processing. Results from the server are returned to the client. □HTLC actions, such as Exec and Call, can initiate processing on an □HTLS server.

All □HTLC client actions initiated by one client are synchronous, except for the CallAsync action, so that the client waits for a response from the server until the server satisfies the request. The wait time is limited to the programmer-selected □HTLC 'Timeout'. An exception is thrown if the timeout is exceeded.

Results provided to the client by the server include two or three elements:

- hasError (bool): 0/No error 1/error occurred
- errMsg (character vector) Empty unless an error occurred
- result value, if any, depends on the operation the client request from the server

The CallAsync action does not block the client processing, so that multiple CallAsync requests can be made to the server without waiting for the results. The CallAsync action responds immediately with an acknowledgement that the server has received the request. The CallAsync result from the server-side is received by the client asynchronously as a callback. The client specifies the APL64 programmer-defined function which will handle the callback. Each CallAsync action has a programmer-provided request identifier.

Server-side Debugging Supported

Both the client and server can be independently debugged when they are based on APL64 Developer version instances.

Simple Example

In this example an APL64 Developer version instance is the client. The client specifies the url for listening. The client uses the □Htlc 'ListenLocal' action to create another APL64 Developer version instance listening on the url. The local server is for the exclusive use of the client. The client uses the □Htlc 'Exec' action to perform a calculation on the server and obtains the result value. The client and server display the operations in their respective history panes, which exist because both the client and server are APL64 Developer version instances. The client-specified action on the server may produce a result, depending on the action.

Results from the server are received by the client as a two- or three-element APL64 vector: (hasErr(bool) errMsg(text) resultValue)

```

□HtlcUrl←'http://localhost:65001'
□Htlc 'ListenLocal'
X←□Htlc 'Exec' '3-15'
0≡1⇒X ∅ No exception
''≡2⇒X ∅ No exception message
3⇒X ∅ Result of server-side calculation

```

The first screenshot shows the APL64: CLEAR WS interface with the following code and output:

```

0 □HtlcUrl←'http://localhost:65001'
1 □Htlc 'ListenLocal'
2 0 □HTLC: Local HTL server process started for listening on http://localhost:65001/
3 X←□Htlc 'Exec' '3-15'
4 0≡1⇒X ∅ No exception
5 1
6 ''≡2⇒X ∅ No exception message
7 1
8 3⇒X ∅ Result of server-side calculation
9 2 1 0 -1 -2
10

```

The second screenshot shows the APL64: CLEAR WS interface with the following code and output:

```

0 '605066b6-5b9f-49de-a1f7-dd1b8e6d7982' □HTLS 'Listen' 'http://localhost:65001/'
1 □HTLS: Server listening on http://localhost:65001/
2 >[□HTLS: EXEC;ExecRes] 3-15
3 2 1 0 -1 -2

```

Local Exclusive or Public Server

If the server being accessed by the client was created by the client using the `□HTLC ListenLocal` action, that server is for the exclusive use of that client. Requests from other clients made to that server are refused.

If the server being accessed by the client was created from an independent APL instance using the `□HTLS Listen` action, that server is a public server which can be accessed by any client via this server's listened url(s).

Synchronous □HTLC actions from the same Client

All □HTLC actions from the same client, except for CallAsync, are synchronous for that client instance of APL64. The client requests server-side processing and waits for result from server. Only the server is listening on a url.

Asynchronous □HTLC CallAsync action

For the CallAsync action the client requests server-side processing, the server acknowledges the request, and client can continue with additional actions.

When server completes the processing request, the server calls back to the client with the result.

The client and server are listening on separate urls, one url for the client to send the request to the server, and one for the server to send the result to the client.

Requests and Results associated with one Client

When the server, listening to one or more urls, receives a request from a client, the response to the client from the server will be returned over the url used by the client to access the server. No other clients will receive this response.

Multiple Clients accessing the same Server

When multiple clients make requests to the same server, it is the responsibility of the APL64 programmer to assure that client requests should represent independent processing requests, with no dependencies between other requests.

Requests by multiple clients to the same server are not queued by that server.

As client requests are satisfied by a server, the state of the current workspace on that client is updated. Subsequent client requests to that server will access that workspace state.

If multiple clients are accessing the same server, the responses from the server to these clients are not necessarily returned in the same order that the requests were received by the server.

Processing time for a request determines when a response can be returned by the server to the requesting client.

A □HTLS server uses separate asynchronous tasks (with OS-assigned independent threads) to process client requests, which do not provide for a guaranteed processing order.

□ HTLC Actions

The □ HTLC system function is used on a client instance of APL64. □ HTLC actions are executed on the server instance of APL64.

Syntax: [listenUrl] □ HTLC Action [arg1] ...

The listenUrl is an optional □ HTLC argument. □ HtlcUrl system variable can be set to contain the required URL value, instead of specifying that URL as a left argument to □ HTLC actions which require a URL value.

Call

The Call action executes a user-defined function on the HTLS server.

Syntax: (hasErr errMsg result) ← [serverUrl] □ HTLC 'Call' aplFnName [rarg] [larg]

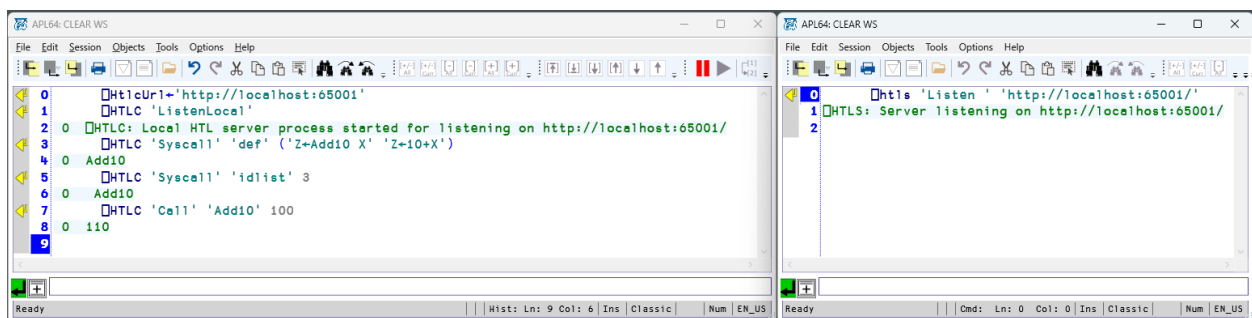
The aplFnName parameter is required. It specifies the name of the APL function to be called. The rarg and larg parameters are optional. Omit both parameters to make a niladic function call. Specify Argument but omit larg to make a monadic function call. Specify both parameters to make a dyadic function call. You cannot specify larg if you omit rarg.

The Call action returns a three-element APL64 vector:

(hasErr(bool) errMsg(text) resultValue)

Example #1: Without Stop in the server-side function

```
'http://localhost:65001' □ HTLC 'ListenLocal'  
'http://localhost:65001' □ HTLC 'Syscall' 'idlist' 3  
'http://localhost:65001' □ HTLC 'Syscall' 'def' ('Z←Add10 X' 'Z←10+X')  
'http://localhost:65001' □ HTLC 'Syscall' 'idlist' 3  
'http://localhost:65001' □ HTLC 'Call' 'Add10' 100
```

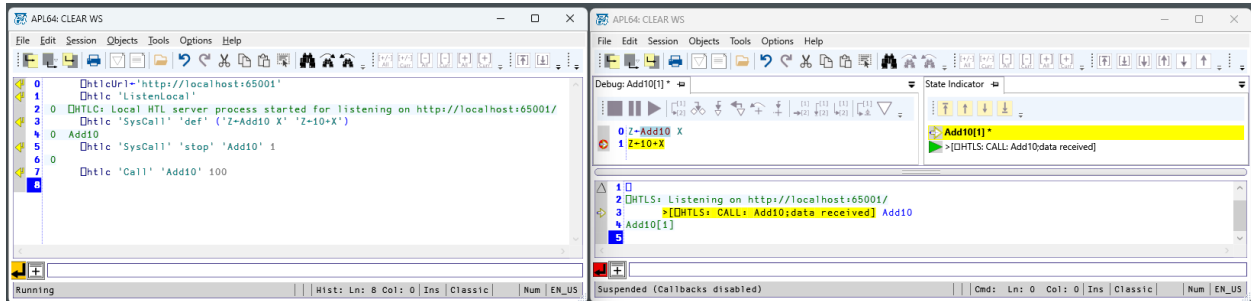


Example #2: With stop in the server-side function:

```

☐ htlcUrl←'http://localhost:65001'
☐ HTLC 'ListenLocal'
☐ HTLC 'SysCall' 'DEF' ('Z←Add10 X' 'Z←10+X')
☐ HTLC 'SysCall' 'Stop' 'Add10' 1
☐ HTLC 'Call' 'Add10' 100

```



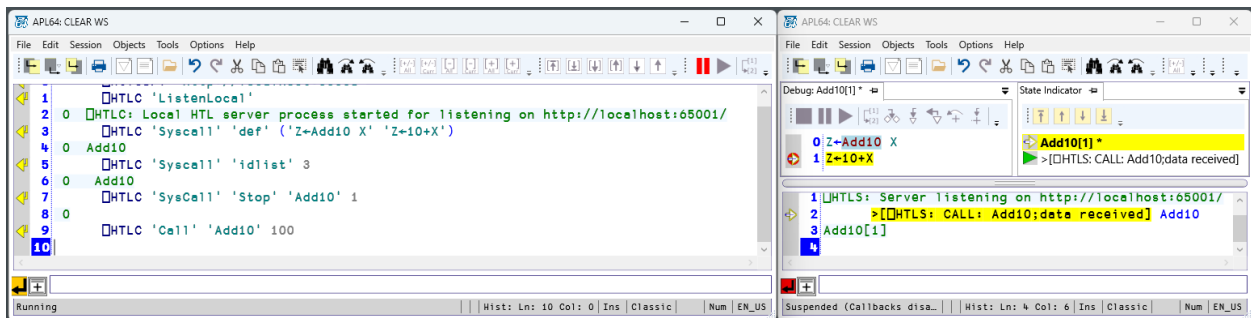
Example #3: With stop in the server-side function, and executing a server-side function directly on the server-side:

```

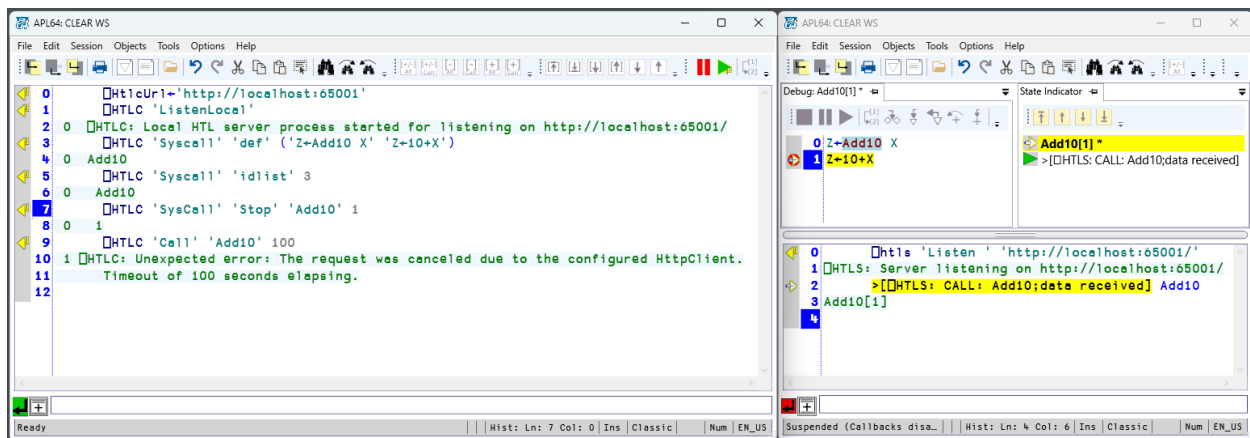
☐ htlcUrl←'http://localhost:65001'
☐ htlc 'ListenLocal'
☐ htlc 'SysCall' 'def' ('Z←Add10 X' 'Z←10+X')
☐ htlc 'SysCall' 'stop' 'Add10' 1
☐ htlc 'Call' 'Add10' 100

```

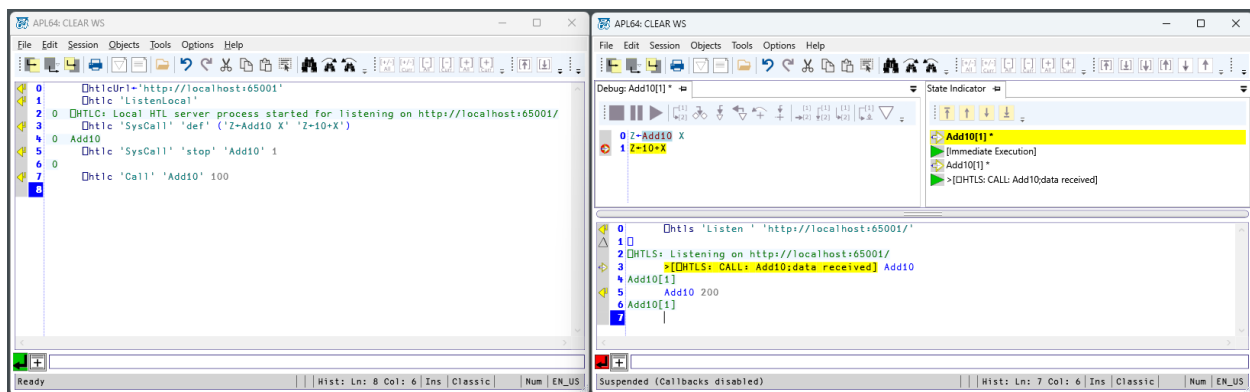
A suspension occurs on the server-side, and the client-side is waiting for a response from the server-side:



In this example if the server-side suspension persists, the client-side timeout will be exceeded. The expiration of the client-side timeout does not modify the server-side state:



In this example after executing the client-side statements, directly execute Add10 200 on the server-side. The server-side state now includes the two pending Add10 execution requests.



CallAsync

The CallAsync action improves application performance in asynchronous execution of a server-side function when many such operations are necessary. Such processing requests can be spread among multiple ☐HTLS servers.

Syntax: (hasErr errMsg result) ← [serverUrl] ☐HTLC 'CallAsync' reqId rUrl cFn sFn [rArg] [lArg]

reqId is a text value which uniquely identifies a CallAsync action. The reqId is returned to the client in the server responses to the client's CallAsync action.

rUrl is the response url on which the client is listening. It is the responsibility of the client to have an APL64 instance listening on the rUrl. Generally, the APL64 instance listening on the rUrl is the client instance, in which case use the ☐HTLS 'Listen' rUrl action to start the client listing on the rUrl.

cFn is the name of the APL64 programmer-defined function which will be run on the APL64 instance listening on the rUrl. Generally, this APL64 instance is the client APL64 instance. The cFn should have no result, no left argument, and a required right argument.

cFn rArg Elt#	Data Type	Description of cFn rArg Elements
1	bool	hasErr
2	Char[]	errMsg
3	(reqId sFnResult)	hasErr is false
3	reqId	hasErr is true

sFn is the name of the server-side APL64 programmer-defined function which will be run on the server side. The sFn may have an optional result.

rArg and lArg are the optional arguments of the sFn on the server side.

The CallAsync action is a symmetric action, because both the client and server machines assume client and server roles.

Before the  HTLC CallAsync action is used:

- The server must be listening on a url
- The client must be listening on a different url

When the CallAsync action is used, the server immediately sends an acknowledgement back to the client indicating that the CallAsync processing request has been received. This design allows the client to continue sending additional CallAsync processing requests to a server, while the previously sent CallAsync requests are being processed by that server.

All CallAsync actions must represent independent calculation requests on the server as there is no guarantee of the order in which CallAsync processing requests will be fully satisfied by the server.

The server's fulfillment of a CallAsync request occurs when the actual result of running the sFn is sent as an  HTLC Call request to the rUrl to run the cFn.

If the client and server are running within instances of the APL64 development version and the server is visible, it is possible to debug a CallAsync action.

Example:

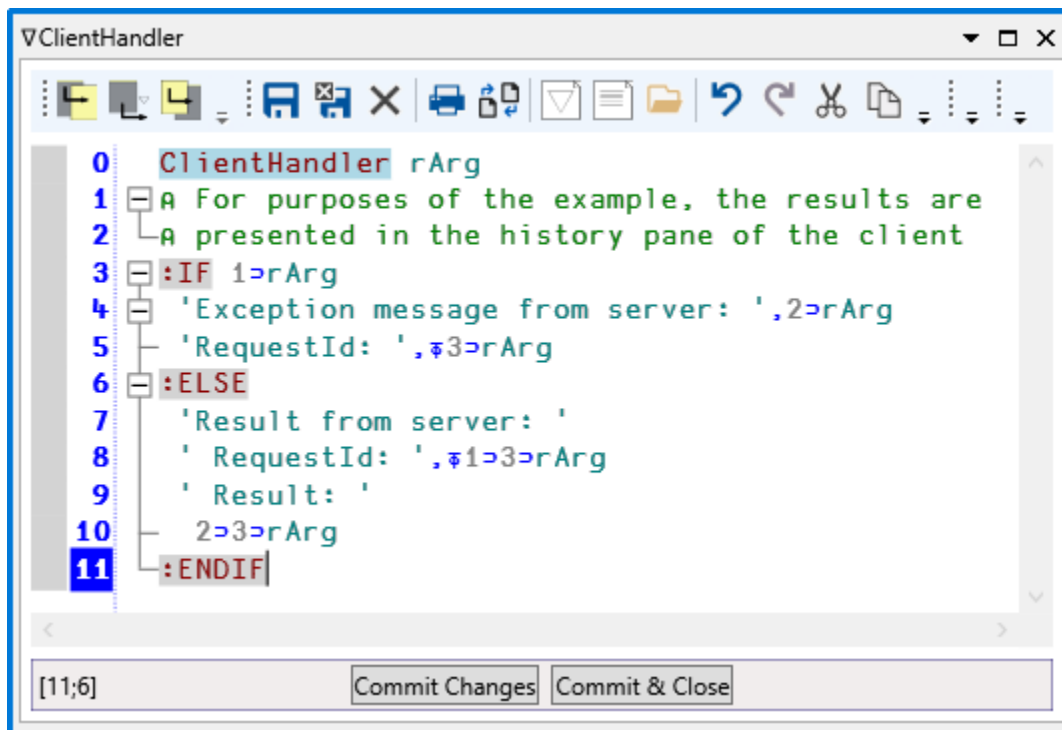
In this example the client will use the CallAsync action to run a function on the server. The client defines the handler function which will handle the asynchronous output from the server and start the client listening on the response url.

ClientHandler rArg
 For purposes of the example, the results are
 presented in the history pane of the client
:IF 1⊃rArg

```

'Exception message from server: ',2>3rArg
'RequestId: ',⌕ 3>3rArg
:ELSE
'Result from server: '
'RequestId: ',⌕ 1>3>3rArg
'Result: '
2>3>3rArg
:ENDIF

```



```

0 ClientHandler rArg
1 A For purposes of the example, the results are
2 A presented in the history pane of the client
3 :IF 1>3rArg
4 'Exception message from server: ',2>3rArg
5 'RequestId: ',⌕ 3>3rArg
6 :ELSE
7 'Result from server: '
8 ' RequestId: ',⌕ 1>3>3rArg
9 ' Result: '
10 2>3>3rArg
11 :ENDIF

```

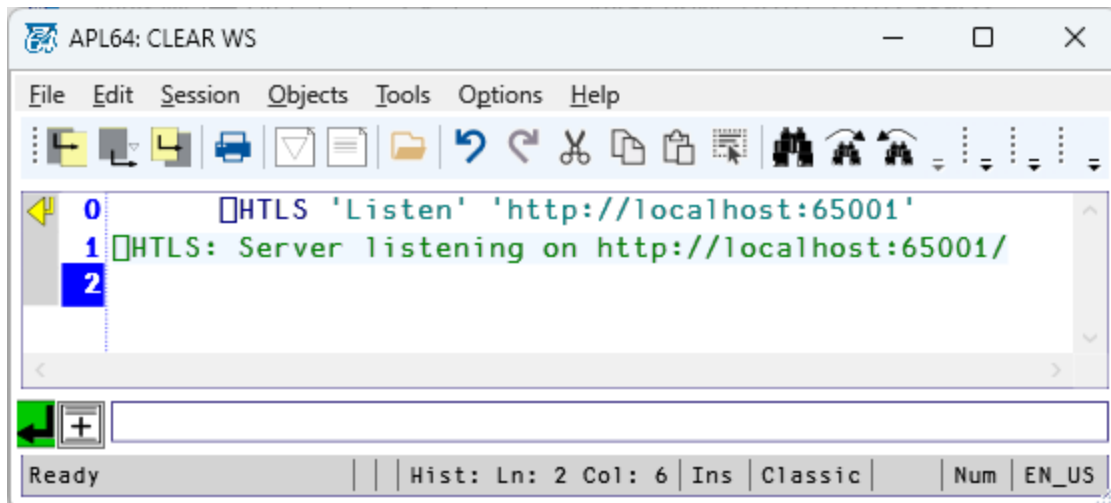
[11;6] Commit Changes Commit & Close

Start the client listening on a specified url, so the client can receive the asynchronous result from the server.

```

☐ HTLS 'Listen' 'http://localhost:65001'

```

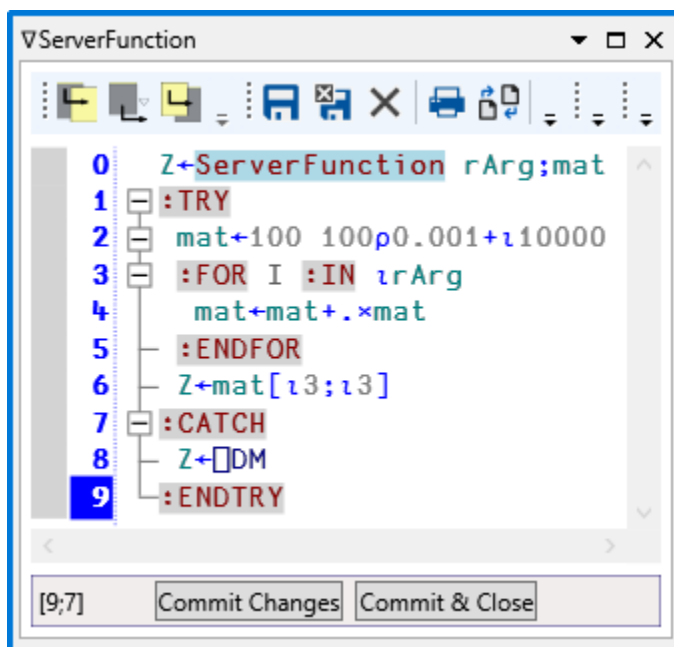


On the server create the function which will be run by the clients CallAsync action:

```

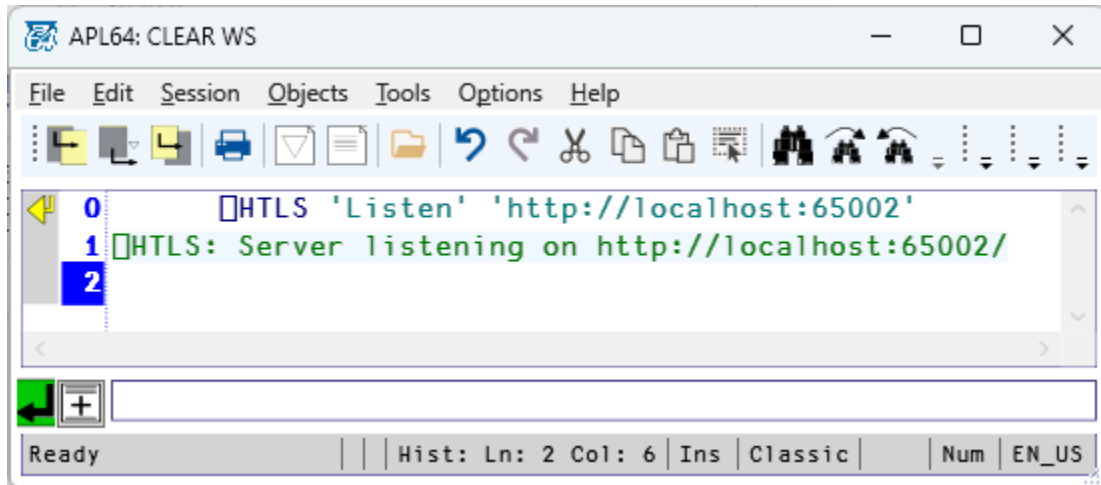
Z←ServerFunction rArg;mat
:TRY
  mat←100 100p0.001+ι10000
  :FOR I :IN rArg
    mat←mat+.×mat
  :ENDFOR
  Z←mat[ι3;ι3]
: CATCH
  Z←⊞DM
:ENDTRY

```



Start the server listening on a different url, so the server can receive the client's CallAsync request.

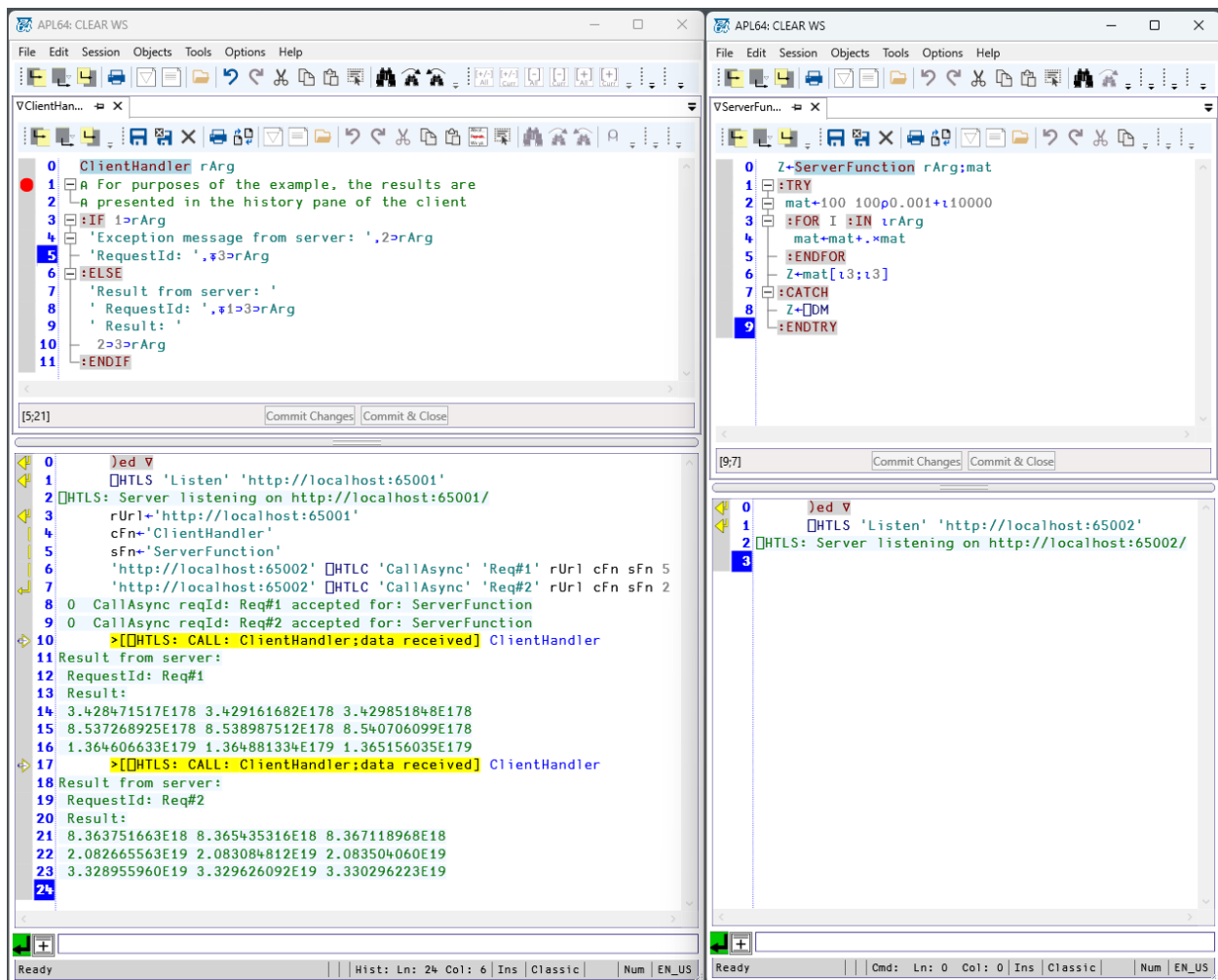
```
☐ HTLS 'Listen' 'http://localhost:65002'
```



On the client, run multiple CallAsync actions as a multi-line statement from the APL64 command line:

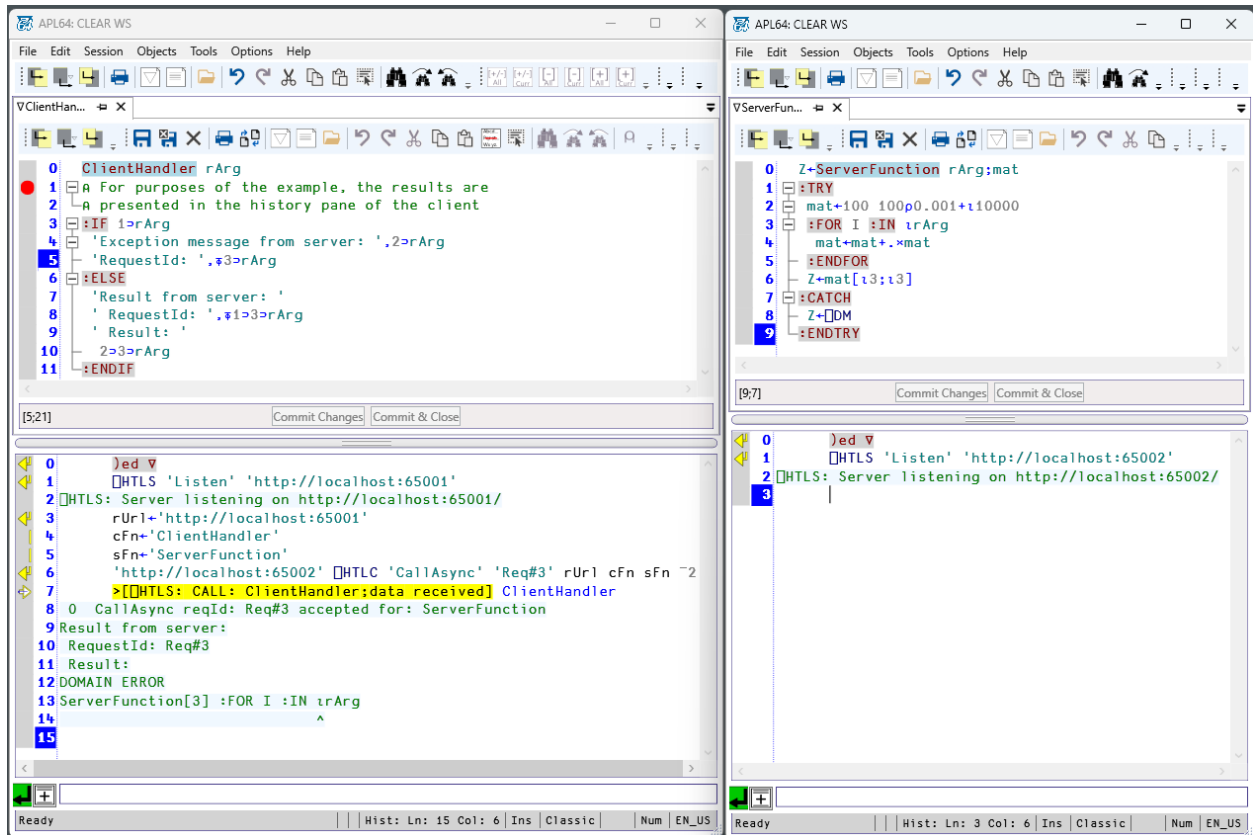
```
rUrl←'http://localhost:65001'
cFn←'ClientHandler'
sFn←'ServerFunction'
'http://localhost:65002' ☐ HTLC 'CallAsync' 'Req#1' rUrl cFn sFn 5
'http://localhost:65002' ☐ HTLC 'CallAsync' 'Req#2' rUrl cFn sFn 2
```

Both client ☐ HTLC CallAsync requests are immediately accepted for processing by the ☐ HTLS server. The results of running the ServerFunction on the server are reported back to the client as asynchronous callbacks. These results are sent from the server as ☐ HTLC Call requests to the client which is running as an ☐ HTLS server:



On the client, make an ☐ HTLC CallAsync request which will trigger an exception in the ServerFunction to see the exception reported from the server to the client via an ☐ HTLC Call action:

'http://localhost:65002' ☐ HTLC 'CallAsync' 'Req#3' rUrl cFn sFn ~2



Exec

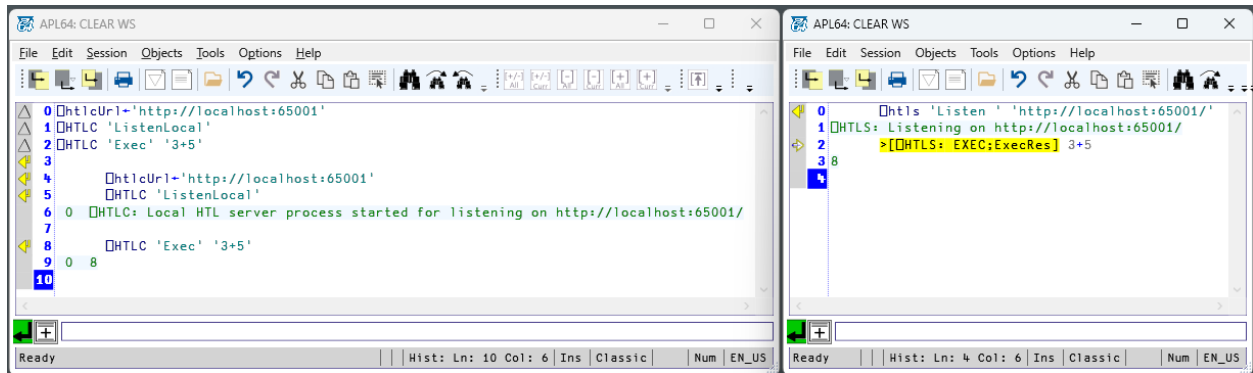
The Exec action executes an APL statement and returns its result value on the HTLS server.

Syntax: (hasErr errMsg result) ← [serverUrl] ☐ HTLC 'Exec' execStmt

execStmt may be a character scalar, character vector or string scalar.

Example #1:

```
☐ htlcUrl←'http://localhost:65001'
☐ HTLC 'ListenLocal'
☐ HTLC 'Exec' '3+5'
```

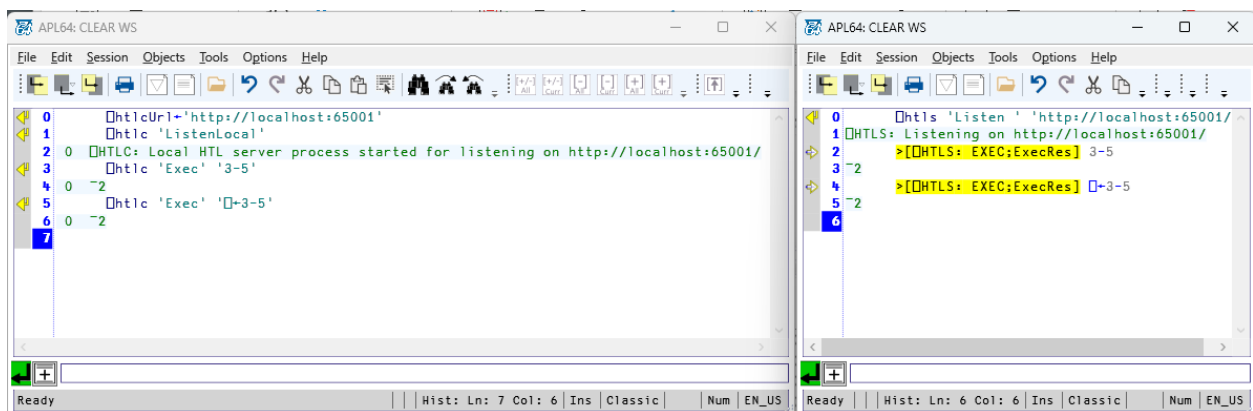


Example #2:

```

⌈htlcUrl←'http://localhost:65001'
⌈htlc 'ListenLocal'
⌈htlc 'Exec' '3-5'
⌈htlc 'Exec' '⌈←3-5'

```

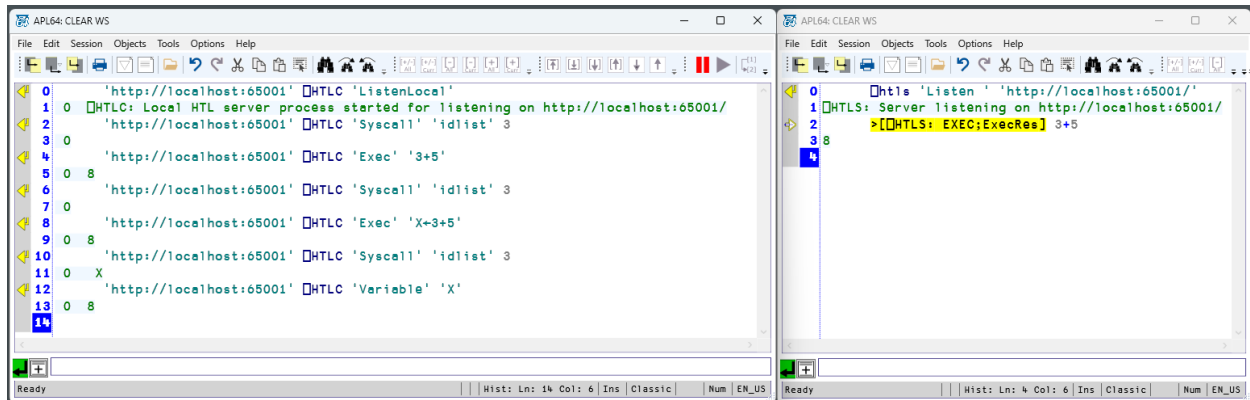


Example #3:

```

'http://localhost:65001' ⌈HTLC 'ListenLocal'
'http://localhost:65001' ⌈HTLC 'Syscall' 'idlist' 3
'http://localhost:65001' ⌈HTLC 'Exec' '3+5'
'http://localhost:65001' ⌈HTLC 'Syscall' 'idlist' 3
'http://localhost:65001' ⌈HTLC 'Exec' 'X←3+5'
'http://localhost:65001' ⌈HTLC 'Syscall' 'idlist' 3
'http://localhost:65001' ⌈HTLC 'Variable' 'X'

```



GetClientId

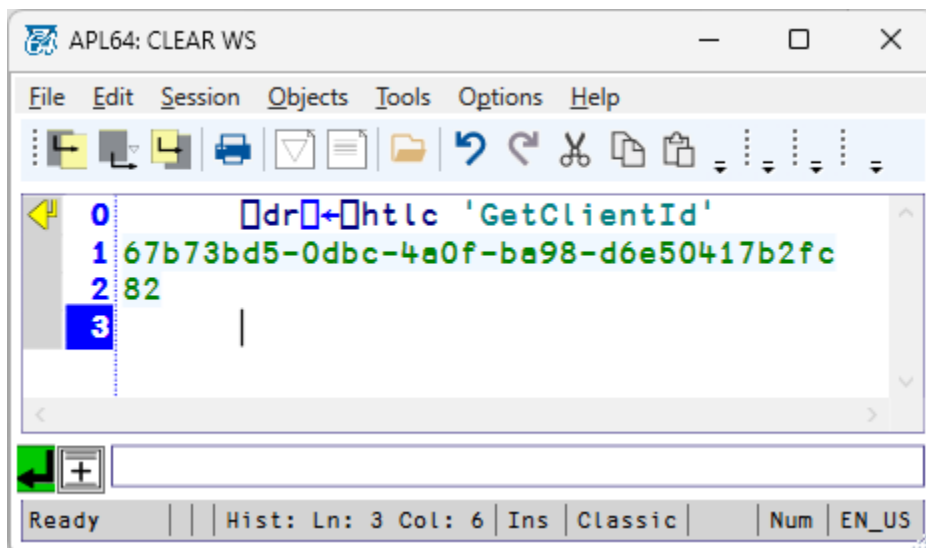
The GetClientId action returns a value used to identify the HTLC client and maintain exclusivity of local HTLS servers.

Syntax: `clientId ← [HTLC 'GetClientId']`

The clientId is a character vector. This value is set the first time an HTLC client action is performed and is constant thereafter for the duration of the client's APL64 instance. Sharing a clientId could compromise the security of a local, exclusive HTLS server.

Example:

```
[dr]←[htlc 'GetClientId']
```



GetServerLog

The GetServerLog action returns the portion of an HTLS server log which is associated with the client.

Syntax: (hasErr, errMsg, [res]) ← [serverUrl] □ HTLC 'GetServerLog'

[res] is the HTLC-specific portion of the csv-format content of the HTLS server log for the HTLS server listening on the serverUrl.

For an example, see the HTLS GetLog action example in this document.

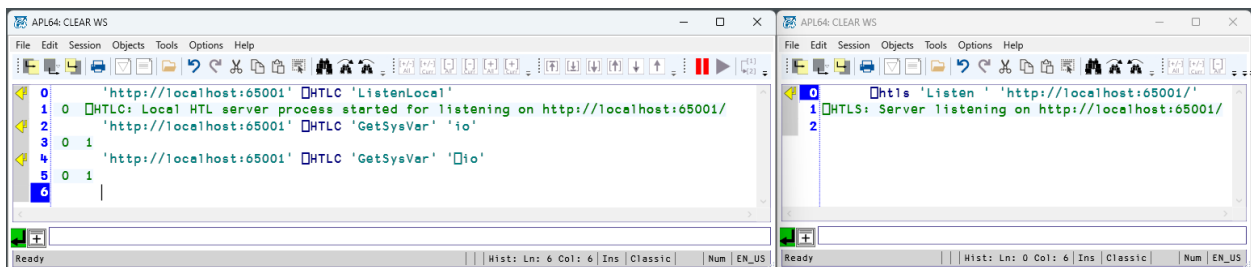
GetSysVar

The GetSysVar action returns the value of the specified system variable on the HTLS server.

Syntax: (hasErr errMsg result) ← [serverUrl] □ HTLC 'GetSysVar' sysVarName

sysVarName may be a character scalar, character vector or string scalar. When specifying the system variable name, the quad (□) prefix is optional.

```
'http://localhost:65001' □ HTLC 'ListenLocal'  
'http://localhost:65001' □ HTLC 'GetSysVar' 'io'  
'http://localhost:65001' □ HTLC 'GetSysVar' '□io'
```



GetVar

The GetVar action returns the programmer-defined variable in the server on the HTLS server.

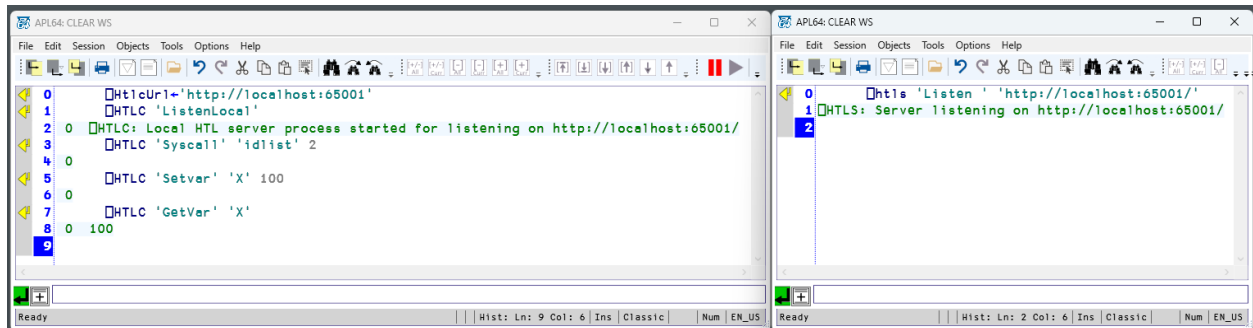
Syntax: (hasErr errMsg result) ← [serverUrl] □ HTLC 'GetVar' varName

varName may be a character scalar, character vector or string scalar.

Example:

```
□ HtlcUrl←'http://localhost:65001'  
□ HTLC 'ListenLocal'  
□ HTLC 'Syscall' 'idlist' 2  
□ HTLC 'Setvar' 'X' 100
```

☐ HTLC 'GetVar' 'X'



Help (?)

The Help action returns the summary documentation of the ☐ HTLC system function.

Syntax: `textMat ← ☐ htlc 'Help'`

`textMat ← ☐ htlc '?'`

```
APL64: CLEAR WS
File Edit Session Objects Tools Options Help
[Icons]
0  []Htlc '?'
1  []HTLC actions:
2
3  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Exec' execStmt
4  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Call' aplFnName [rArg] [lArg]
5  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'CallAsync' reqId rUrl cFn sFn [rArg] [lArg]
6  clientId + []HTLC 'GetClientId'
7  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'GetServerLog'
8  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'GetSysVar' sysVarName
9  (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'GetVar' varName
10 textMat + []htlc 'Help'
11 textMat + []htlc '?'
12 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'ListenLocal' [restartExistingServer] [wsToLoad]
13 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'LoadWs' [wsToLoad]
14 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'ServerOff'
15 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'SetSysVar' sysVarName sysVarValue
16 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'SetVar' varName varValue
17 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Stop'
18 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'SysCall' sysFnName [rArg] [lArg]
19 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'SysCommand' 'sysCmdName rArg'
20 priorMs + [serverUrl] []HTLC 'Timeout' [milliseconds]
21 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Variable' varName
22 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Variable' varName varValue
23 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Visible' [0 or 1]
24 (hasErr, errMsg, [res]) + [serverUrl] []HTLC 'Visible'
25
26 serverUrl: http://serverName:port#
27 clientId: GUID string identifying this client session
28 Set []HtlcUrl instead of specifying serverUrl for the []HTLC left argument
29
30 Note: Stop and ServerOff actions only work on private servers (created
31 with []HTLC 'ListenLocal') where the requesting client owns
32 the server. Public servers cannot be controlled by clients.
33
34 Note: GetServerLog returns only log entries for this client's ID (filtered
35 on the server side). To see all log entries, use []HTLS 'GetLog'
36 directly on the server.
37
38 Note: Pause and Resume actions are server-side only ([]HTLS). Clients cannot
39 pause/resume servers since pausing would prevent further communication.
40
```

wsToLoad is an optional path of the APL64 workspace to load.

restartExistingServer: Optional bool scalar indicating if a request will be made to the specified url to run the server-side 'ServerOff' action prior to sending the request to start the local, exclusive server listening on the url. The restartExistingServer option is effective only if the server to be restarted is exclusively controlled by the APL64 instance which used the ListenLocal action with this option.

The 'wsToLoad' and 'restartExistingServer' options can be included in any order in the ListenLocal statement.

The ListenLocal action:

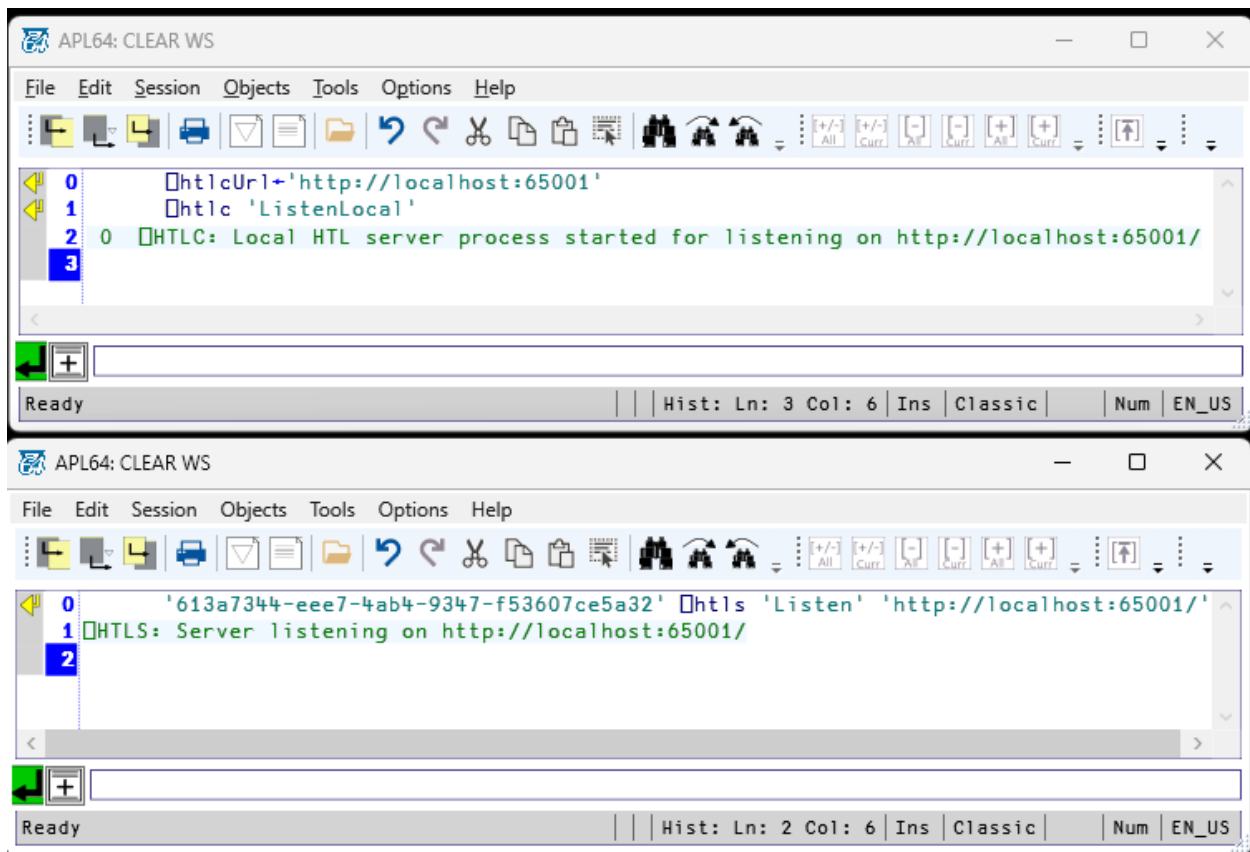
- Uses the .Net [ProcessStart method](#) to create a separate server instance of APL64 and start the server listening for incoming requests
- Starts the server instance of APL64 with the 'HTLListenUrl = serverUrl' command line parameter
- Cannot be used to start a remote server
- Does not support an https url
- Does not support multiple urls for listening
- The ListenLocal action creates a local ☐HTLS server for exclusive use by the client

When the server side, created by the 'ListenLocal' action, attempts to start listening, the url will be validated and any exceptions presented on the server-side.

Example 1: The client uses the ☐HTLC 'ListenLocal' action to create a local ☐HTLS server

```
☐htlcUrl←'http://localhost:65001'  
☐htlc 'ListenLocal'
```

The .Net ProcessStart method is used to start an APL64 instance of the same type as the client. The .Net ProcessStart arguments for the new APL instance include the url and the unique id of the client. The unique id is retained by the server to ensure that the server is for the exclusive use of this client.



Example 2: Server-side debugging

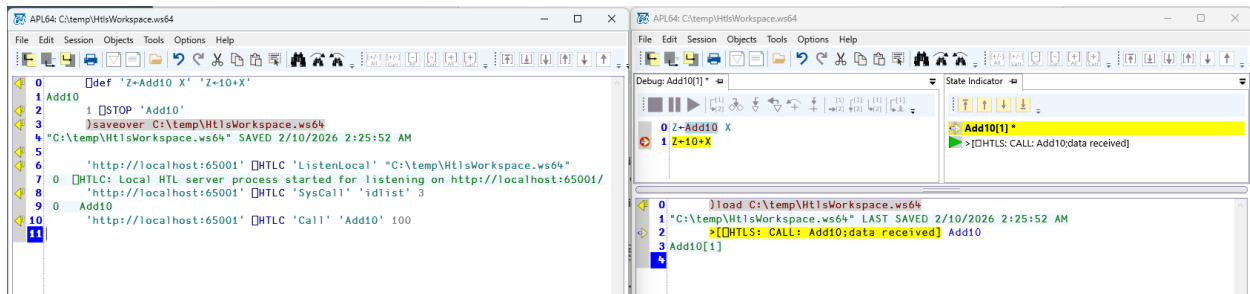
Prepare a workspace containing the function `⎕def 'Z←Add10 X' 'Z←10+X'`, put a stop on line [1] of the function using `1 ⎕STOP 'Add10'`, and save the `c:\temp\HtlsWorkspace.ws64`.

```

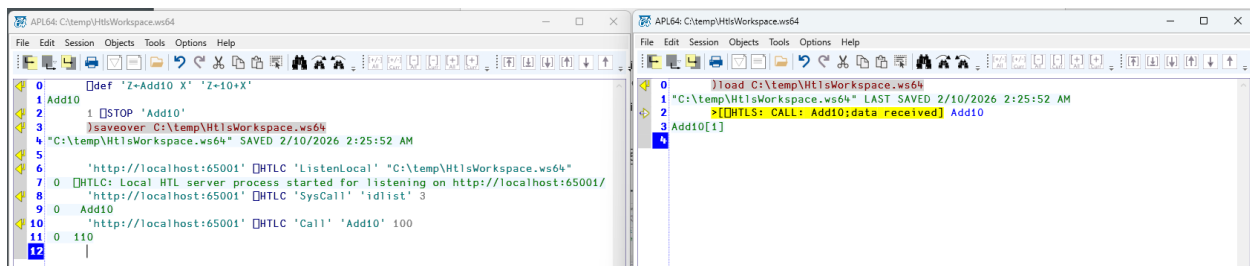
⎕def 'Z←Add10 X' 'Z←10+X'
1 ⎕STOP 'Add10'
)saveover C:\temp\HtlsWorkspace.ws64
'http://localhost:65001' ⎕HTLC 'ListenLocal' "C:\temp\HtlsWorkspace.ws64"
'http://localhost:65001' ⎕HTLC 'SysCall' 'idlist' 3
'http://localhost:65001' ⎕HTLC 'Call' 'Add10' 100

```

When the client calls the Add10 function, that function runs to the stop on the server, and the APL64 debugger pane is presented on the server:



Continue the execution of the Add10 function on the server, and the result will be returned to the client, unless the programmer timeout on the client-side is exceeded.:



Example #3: Server name is not LocalHost

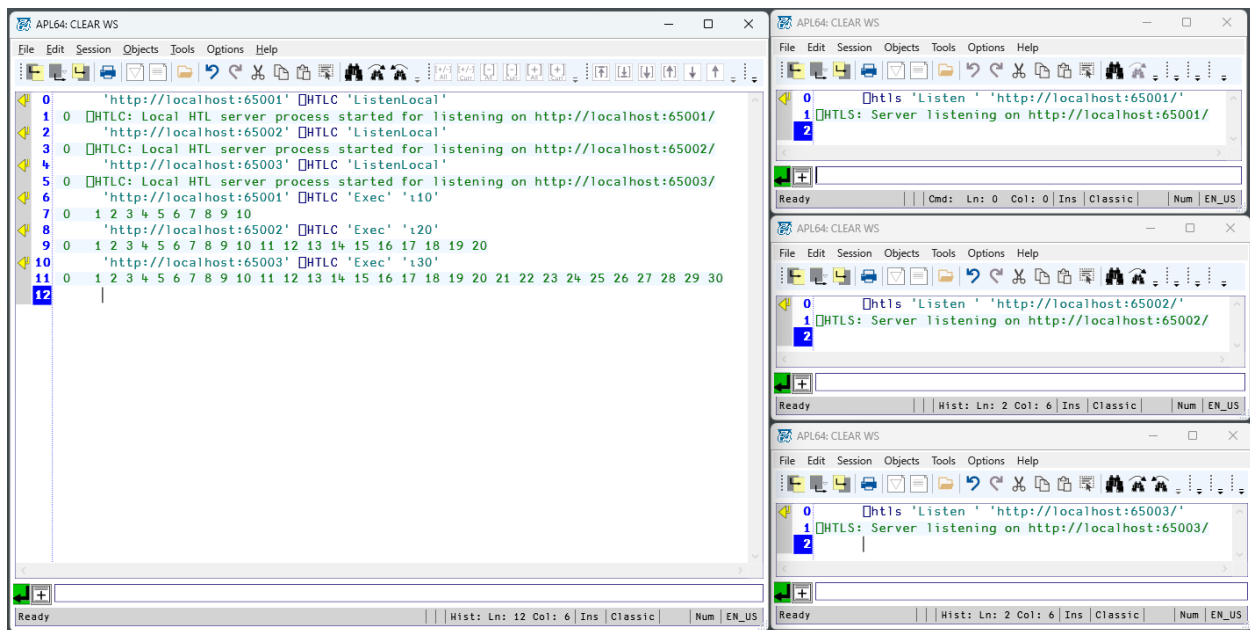
Using an IPv4 address or a server name other than local host:

- Cannot be used with the ☐ HTLC ListenLocal action
- When used with the ☐ HTLS Listen action, administrator access to the server machine to authorize communication with the server and port forwarding may be required

Example #4: Multiple Servers

In this example the ☐ Htlc 'ListenLocal' action is used to create multiple independent servers. Each server can have a different workspace loaded and perform different actions.

```
'http://localhost:65001' ☐ HTLC 'ListenLocal'
'http://localhost:65002' ☐ HTLC 'ListenLocal'
'http://localhost:65003' ☐ HTLC 'ListenLocal'
'http://localhost:65001' ☐ HTLC 'Exec' 'i10'
'http://localhost:65002' ☐ HTLC 'Exec' 'i20'
'http://localhost:65003' ☐ HTLC 'Exec' 'i30'
```



Example #5: Exclusive listening server already exists: Restart it

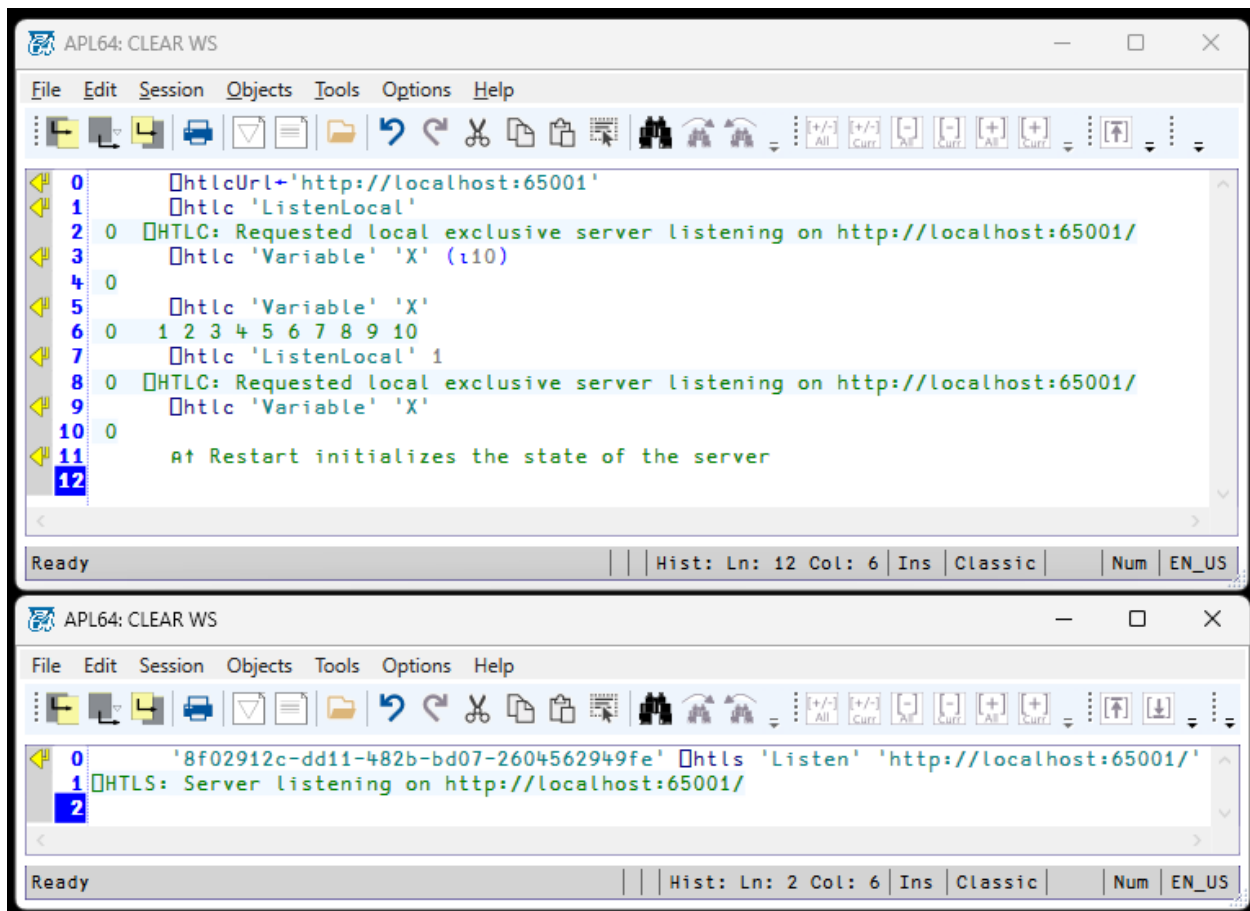
Create an exclusive listening server.

Modify the state of that server.

Use the ListenLocal action with the 'RestartExistingServer' option active.

Observe the server's state is initialized.

- ☐ htclUrl←'http://localhost:65001'
- ☐ htcl 'ListenLocal'
- ☐ htcl 'Variable' 'X' (i10)
- ☐ htcl 'Variable' 'X'
- ☐ htcl 'ListenLocal' 1
- ☐ htcl 'Variable' 'X'
- ☒ ↑ Restart initializes the state of the server



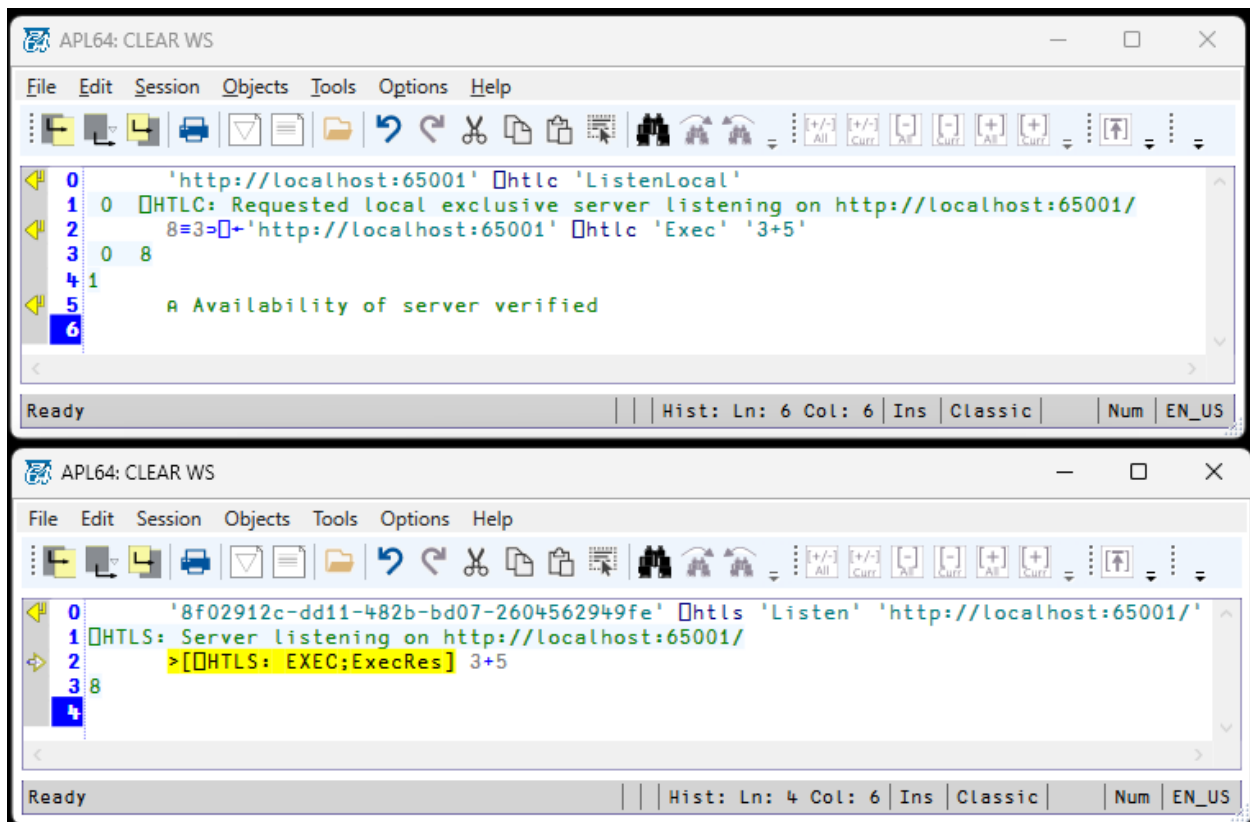
Example #6: Check for a pre-existing server

Create a listening server. Use an Exec action to check if the server is listening. If a valid response is returned, the server is listening. If the timeout expires, the server does not exist or is paused or stopped.

```

'http://localhost' ⎕htlc 'ListenLocal'
8≡3⇒ ⎕←'http://localhost:65001' ⎕htlc 'Exec' '3+5'

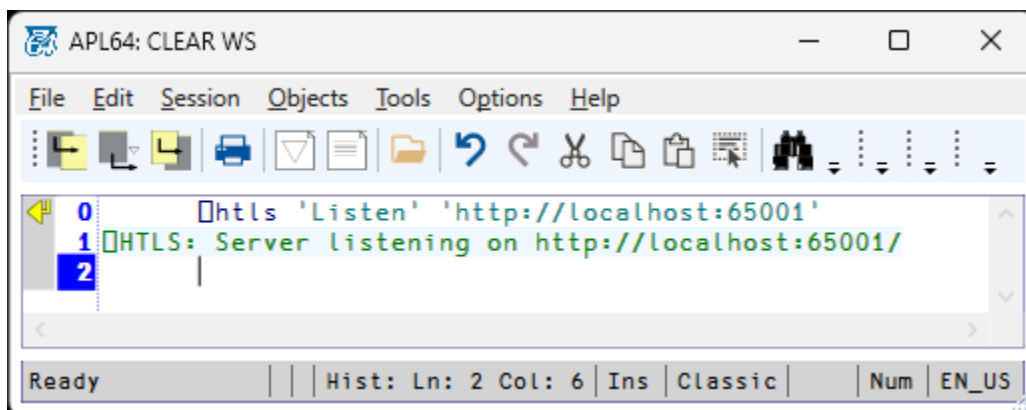
```



Example #7: Listening server already exists: Exception occurs

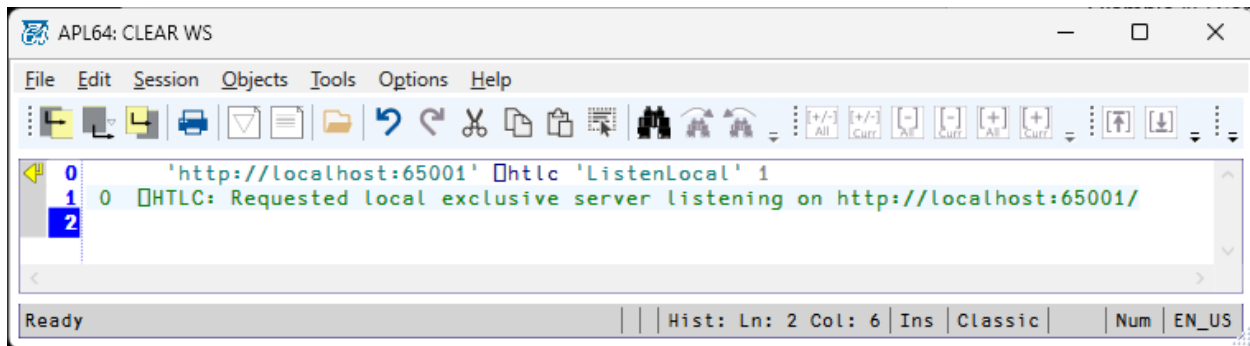
Create a non-exclusive ☐ HTLS server in an APL64 instance:

☐htls 'Listen' 'http://localhost:65001'



In a different APL64 instance on the same workstation attempt to create a server listening on the same url:

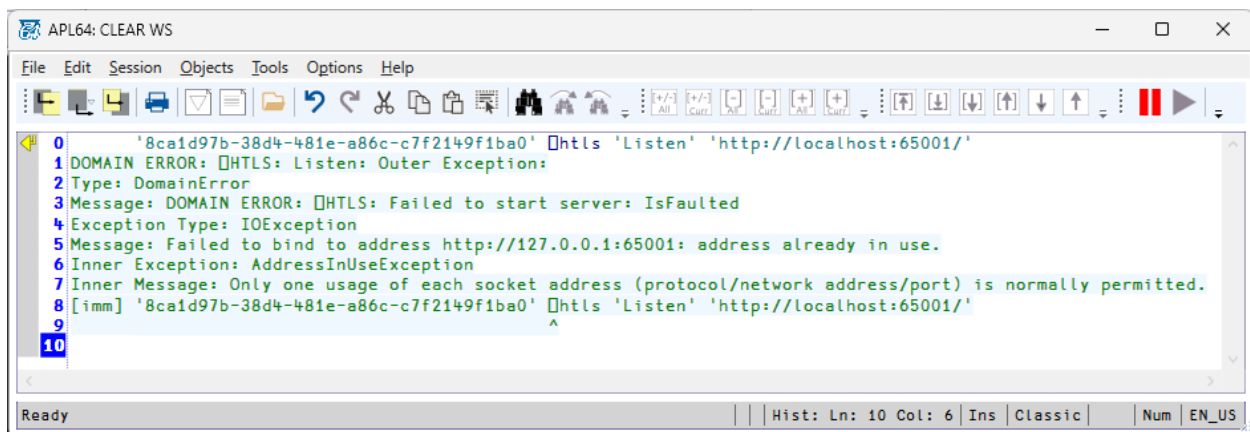
'http://localhost:65001' ☐htlc 'ListenLocal' 1



```
0 'http://localhost:65001' [htlc 'ListenLocal' 1
1 0 [HTLC: Requested local exclusive server listening on http://localhost:65001/
2
```

Ready | Hist: Ln: 2 Col: 6 | Ins | Classic | Num | EN_US

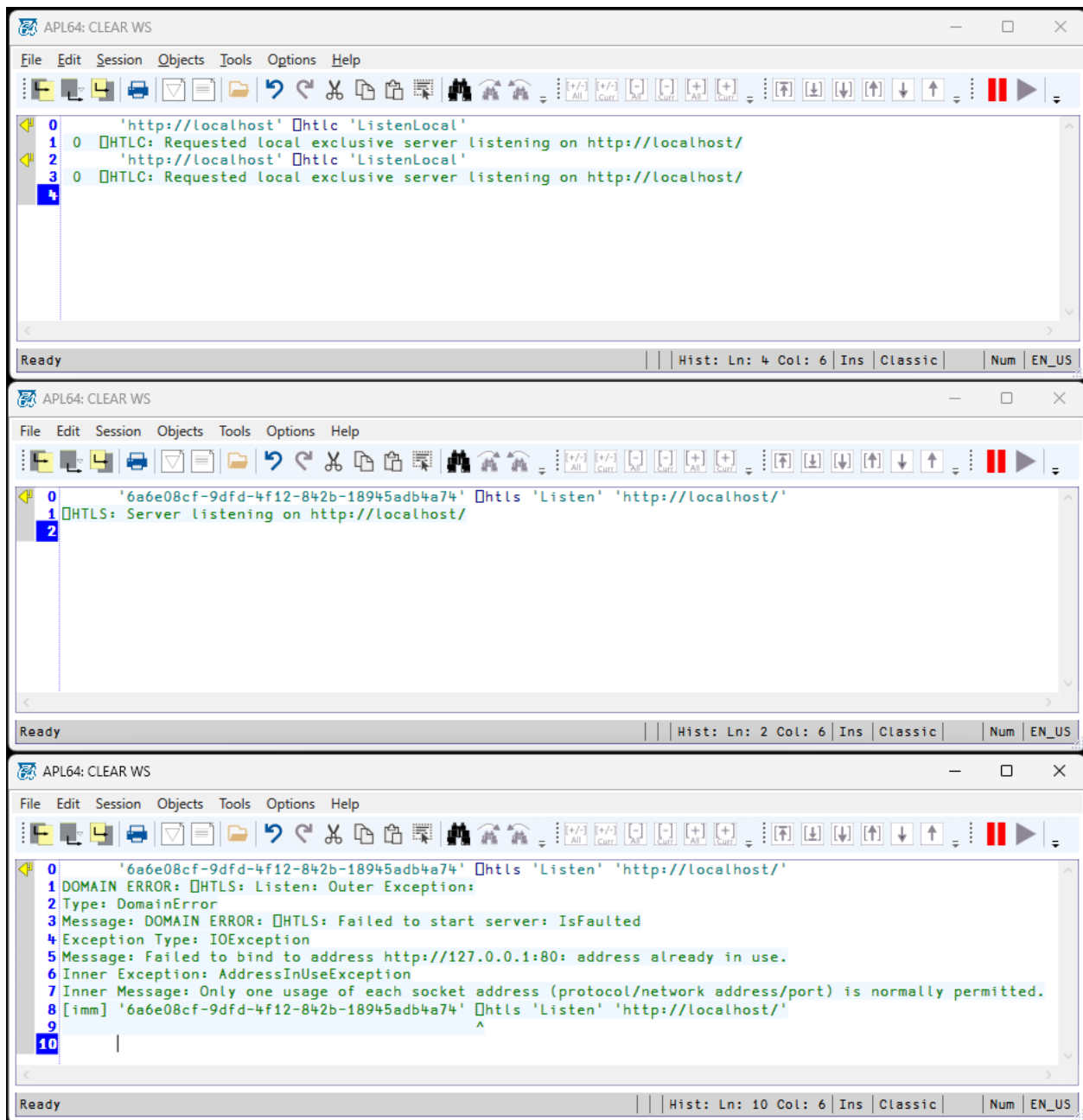
The ListenLocal action will create an APL64 instance, but that instance cannot be listening on the specified url, because a server on the workstation is already listening on that url. An exception is thrown on the new APL64 instance which would not start listening:



```
0 '8ca1d97b-38d4-481e-a86c-c7f2149f1ba0' [htls 'Listen' 'http://localhost:65001/'
1 DOMAIN ERROR: [HTLS: Listen: Outer Exception:
2 Type: DomainError
3 Message: DOMAIN ERROR: [HTLS: Failed to start server: IsFaulted
4 Exception Type: IOException
5 Message: Failed to bind to address http://127.0.0.1:65001: address already in use.
6 Inner Exception: AddressInUseException
7 Inner Message: Only one usage of each socket address (protocol/network address/port) is normally permitted.
8 [imm] '8ca1d97b-38d4-481e-a86c-c7f2149f1ba0' [htls 'Listen' 'http://localhost:65001/'
9 ^
10
```

Ready | Hist: Ln: 10 Col: 6 | Ins | Classic | Num | EN_US

The same exception will occur if the pre-existing server was an exclusive server:



LoadWs

The Load action loads an APL64 workspace in the HTLS server.

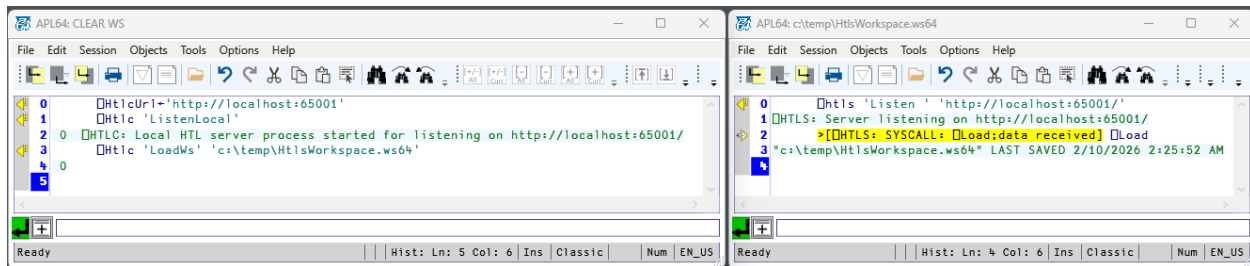
Syntax: (hasErr errMsg result) ← [serverUrl] ⌈HTLC 'LoadWs' ws

ws is the name of the APL64 workspace to load.

Example:

```
⌈HtlcUrl←'http://localhost:65001'
```

- ☐ Htlc 'ListenLocal'
- ☐ Htlc 'LoadWs' 'c:\temp\HtIsWorkspace.ws64'



ServerOff

The ServerOff action requests the server-side APL64 instance to stop listening.

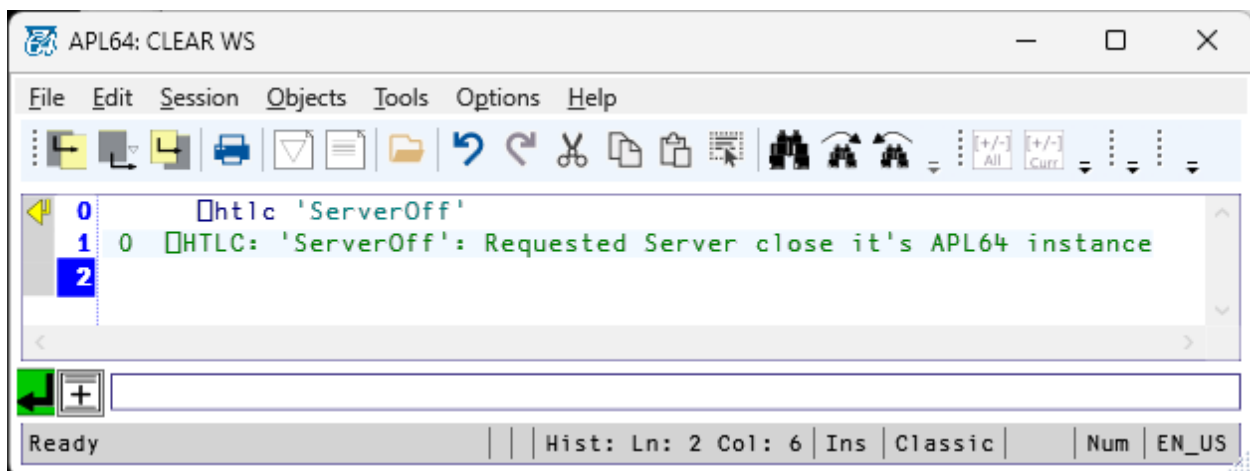
Syntax: (hasErr errMsg result) ← [serverUrl] ☐ HTLC 'ServerOff'

Since a server created by the client using the ListenLocal action, it is an exclusive server which no other clients can access, so the ServerOff action cannot affect other ☐ HTLC clients.

Example:

- HtlcUrl←'http://localhost:65001'

 - ☐ HTLC 'ListenLocal'
 - ☐ HTLC 'ServerOff'



SetSysVar

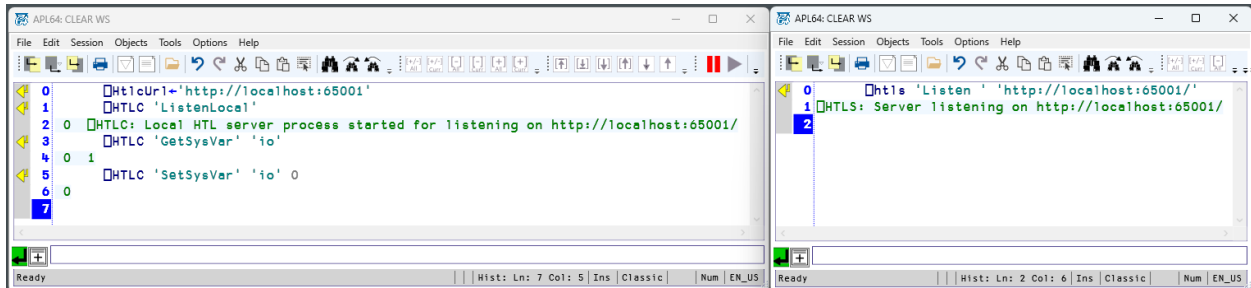
The SetSysVar action sets the value of the specified system variable on the HTLS server.

Syntax: (hasErr errMsg) ← [serverUrl] ☐ HTLC 'SetSysVar' sysVarName sysVarValue

sysVarValue may be a character scalar, character vector or string scalar. When specifying the system variable name, the quad (□) prefix is optional.

Example:

```
□ HtlcUrl←'http://localhost:65001'  
□ HTLC 'ListenLocal'  
□ HTLC 'GetSysVar' 'io'  
□ HTLC 'SetSysVar' 'io' 0
```



SetVar

The SetVar action sets the programmer-defined variable on the HTLS server.

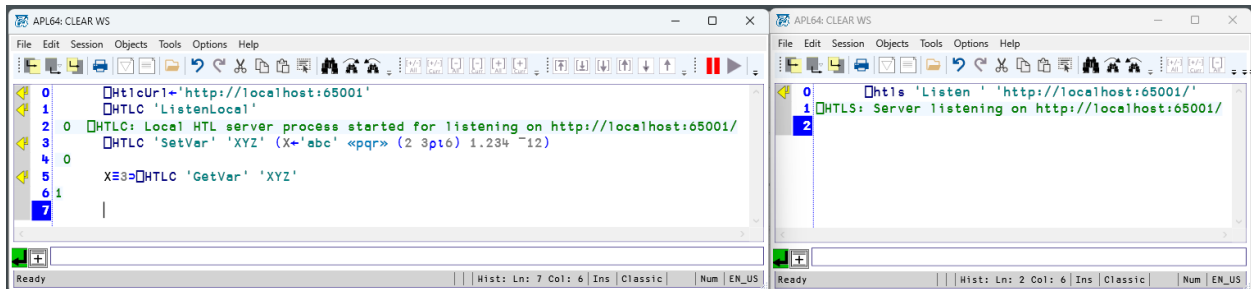
Syntax: (hasErr errMsg) ← [serverUrl] □HTLC 'SetVar' varName varValue

varName is the name of the programmer-defined variable and may be a character scalar, character vector or string scalar. If the specified variable does not exist it will be created.

varValue is the value to assign to the variable.

Example:

```
□ HtlcUrl←'http://localhost:65001'  
□ HTLC 'ListenLocal'  
□ HTLC 'SetVar' 'XYZ' (X←'abc' «pqr» (2 3pt6) 1.234 ~12)  
X≡3⇒□HTLC 'GetVar' 'XYZ'
```



Stop

The Stop action requests the server to stop listening.

Syntax: (hasErr errMsg result) ← [serverUrl] ☐HTLC 'Stop'

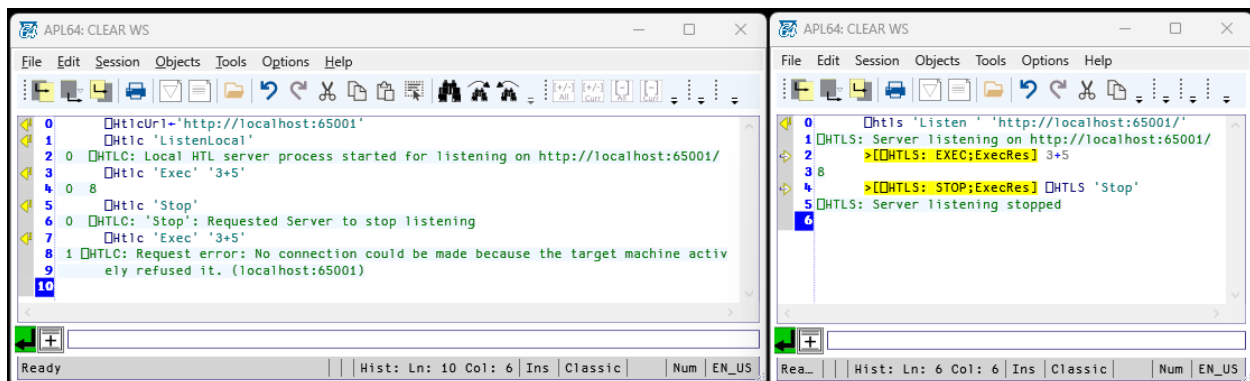
The Stop action stops the server listening on all urls which were configured for that server.

Note: This action may take a few seconds to complete.

The client-side cannot coordinate the Stop action with other instances of APL64 which may be communicating with the server.

Example:

```
☐ HtlcUrl←'http://localhost:65001'  
☐ Htlc 'ListenLocal'  
☐ Htlc 'Exec' '3+5'  
☐ Htlc 'Stop'  
☐ Htlc 'Exec' '3+5'
```



SysCall

The SysCall action calls a system-function on the HTLS server.

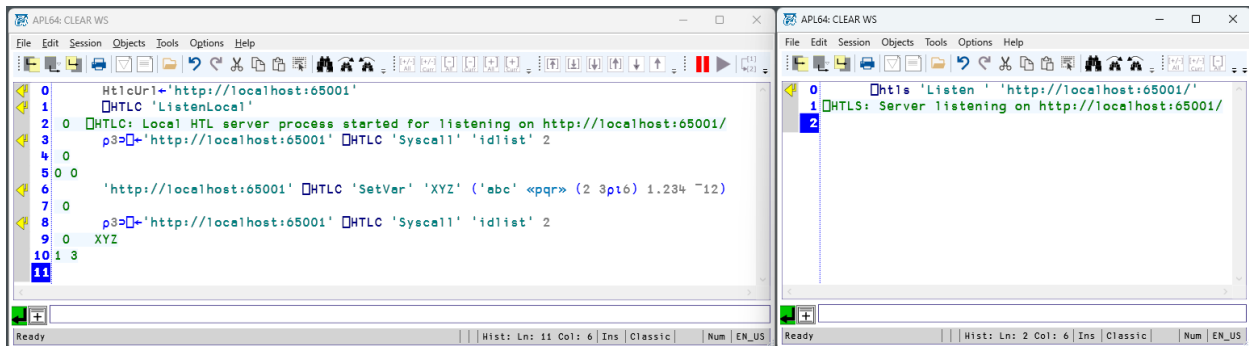
Syntax: (hasErr errMsg result) ← [serverUrl] ☐HTLC 'SysCall' sysFnName [rarg] [larg]

The SysCall action is similar to the Call action. However, it calls a system-function rather than user-defined functions. When the system-function is not prefixed by '☐', it will be added by the SysCall action.

Example:

```
HtlcUrl←'http://localhost:65001'  
☐ HTLC 'ListenLocal'  
p3▷☐←'http://localhost:65001' ☐HTLC 'Syscall' 'idlist' 2
```

```
'http://localhost:65001' □ HTLC 'SetVar' 'XYZ' ('abc' «pqr» (2 3p16) 1.234 ~12)
p3>□←'http://localhost:65001' □ HTLC 'Syscall' 'idlist' 2
```



SysCommand

The SysCommand action executes an APL system command on the HTLS server.

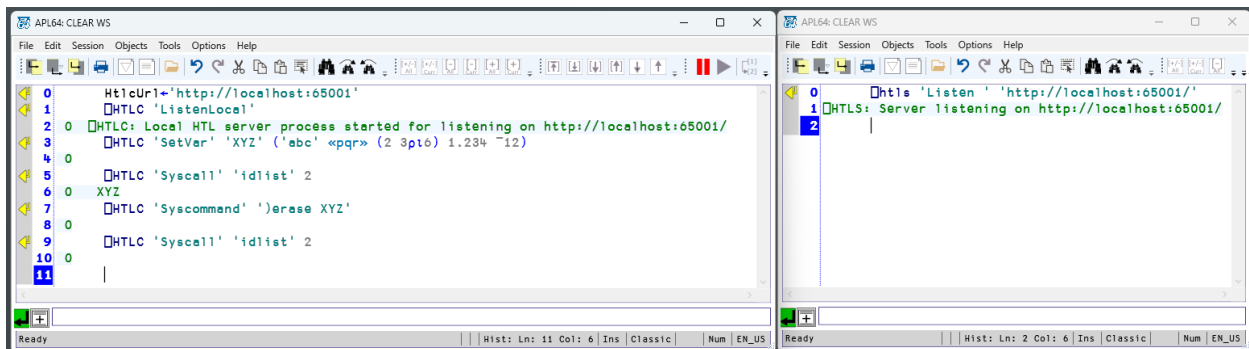
Syntax: (hasErr errMsg) ← [serverUrl] □ HTLC 'SysCommand' 'sysCmdName rArg'

If the programmer-provided 'sysCmdName' is not prefixed by ')', it will be added by the SysCommand' action.

The SysCommand action output includes only the hasErr and errMsg elements, as no result value of a system command is returned to the client-side.

Example:

```
HtclUrl←'http://localhost:65001'
□ HTLC 'ListenLocal'
□ HTLC 'SetVar' 'XYZ' ('abc' «pqr» (2 3p16) 1.234 ~12)
□ HTLC 'Syscall' 'idlist' 2
□ HTLC 'Syscommand' ' )erase XYZ'
□ HTLC 'Syscall' 'idlist' 2
```



Timeout

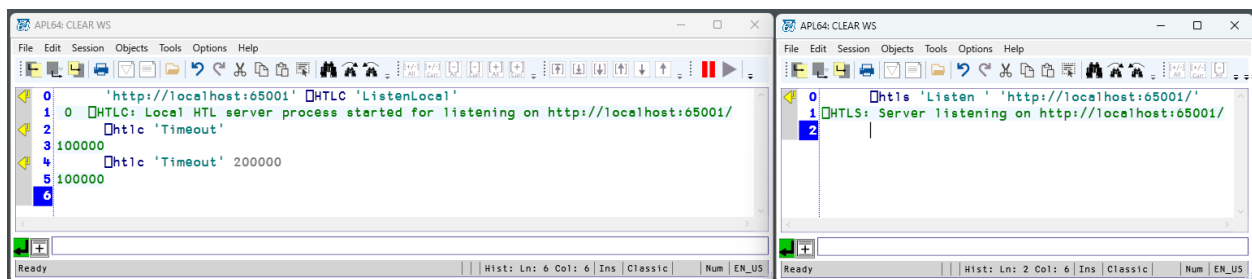
The timeout action sets or returns the length of time the client will wait for a response from the HTLS server.

Syntax: priorMs ← [serverUrl] □HTLC 'Timeout' [milliseconds]

The default value is 100000 milliseconds.

Example:

```
'http://localhost:65001' □HTLC 'ListenLocal'  
□htlc 'Timeout'  
□htlc 'Timeout' 200000
```



Variable

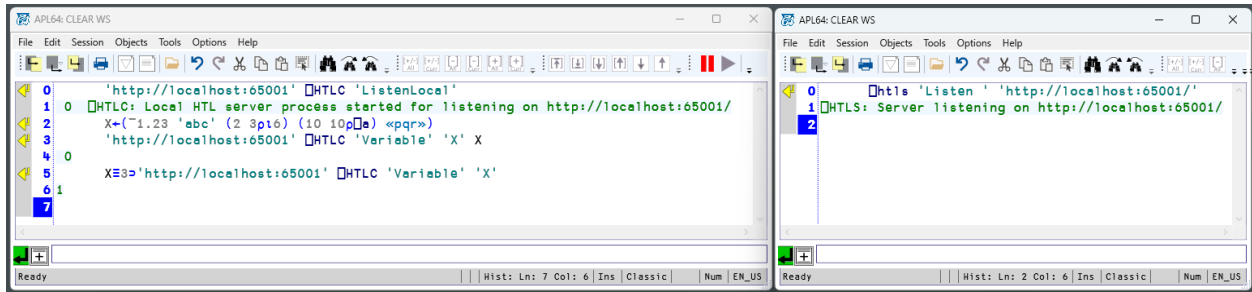
The Variable action sets or returns the value of a user-defined variable on the HTLS server.

Syntax: (hasErr errMsg result) ← [serverUrl] □HTLC 'Variable' varName [varValue]

If the specified variable does not exist it will be created. This is implemented as a property that takes the variable name and optional value as its arguments.

Example:

```
'http://localhost:65001' □HTLC 'ListenLocal'  
X←(¬ 1.23 'abc' (2 3p16) (10 10p□a) «pqr»)   
'http://localhost:65001' □HTLC 'Variable' 'X' X  
X≡3⇒'http://localhost:65001' □HTLC 'Variable' 'X'
```



Visible

The Visible action controls the visibility of the ☐ HTLS server.

Syntax: (hasErr errMsg result) ← [serverUrl] ☐ HTLC 'Visible' flag

flag is the optional argument to control the visibility of the ☐ HTLS server. The default is 1 which means the ☐ HTLS server is visible.

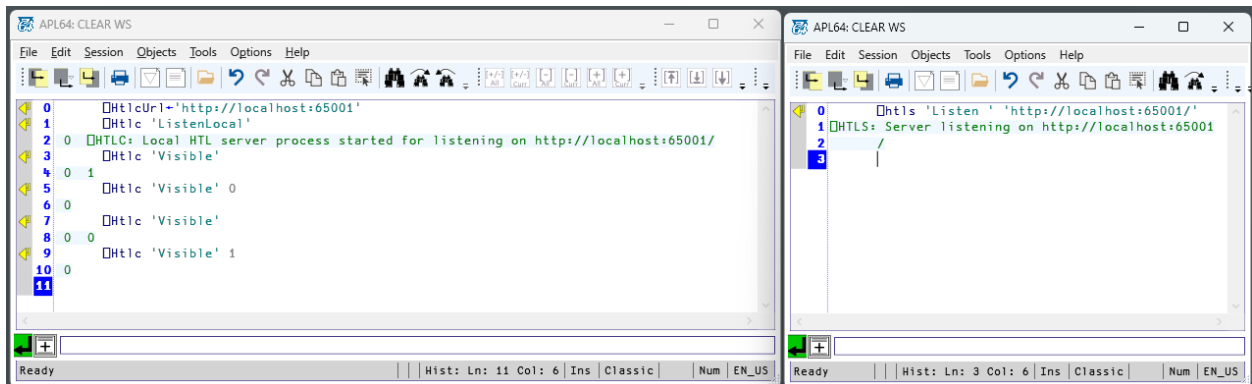
The server can be made visible only when the server is based on an APL64 Developer version instance. When server-side exceptions occur which cannot be transmitted to the client-side, the exception message will be rendered to the history pane of the visible server-side.

Example:

```

☐ HtlcUrl←'http://localhost:65001'
☐ Htlc 'ListenLocal'
☐ Htlc 'Visible'
☐ Htlc 'Visible' 0
☐ Htlc 'Visible'
☐ Htlc 'Visible' 1

```



□ HTLS Actions

The □ HTLS system function is used on the server instance of APL64. □ HTLS actions are performed on the server instance of APL64.

Syntax: [result] ← □ HTLS Action [arg1] ...

The server processing to satisfy client requests is asynchronous using multiple .Net tasks with OS-assigned independent threads.

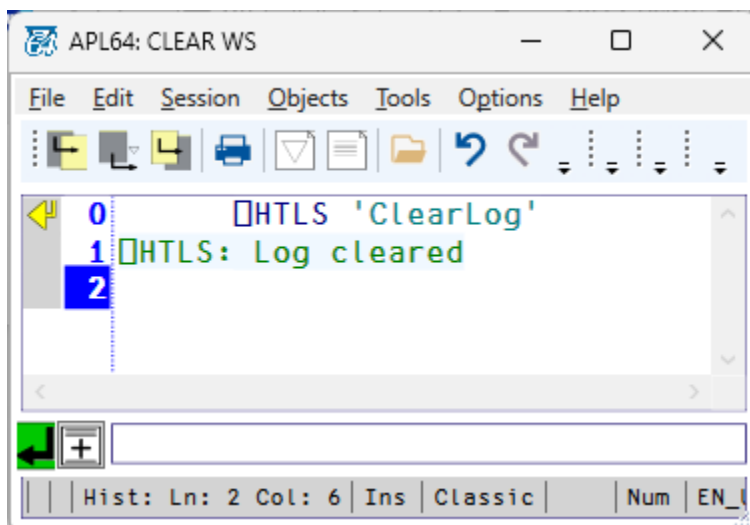
ClearLog

The ClearLog action clears the content of the current log file, if any.

Syntax: □ HTLS 'ClearLog'

Example:

```
□ HTLS 'ClearLog'
```



GetLog

The GetLog action returns the content of the current HTLS log file, if any, in csv format.

Syntax: logLines(csv) ← □ HTLS 'GetLog'

Example:

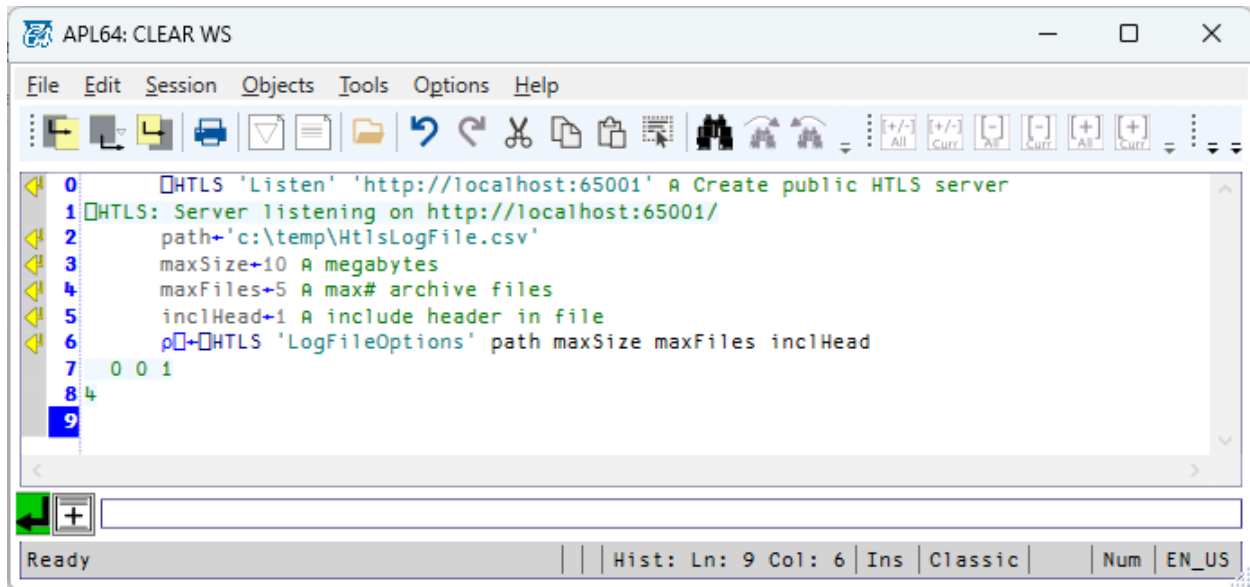
In this example a client tries to stop a public server, which is not permitted.

On an instance of the APL64 Developer version, create a public HTLS server:

```

⊞ HTLS 'Listen' 'http://localhost:65001' ⊞ Create public HTLS server
path←'c:\temp\HtIsLogFile.csv'
maxSize←10 ⊞ megabytes
maxFiles←5 ⊞ max# archive files
inclHead←1 ⊞ include header in file
ρ⊞←⊞ HTLS 'LogFileOptions' path maxSize maxFiles inclHead

```

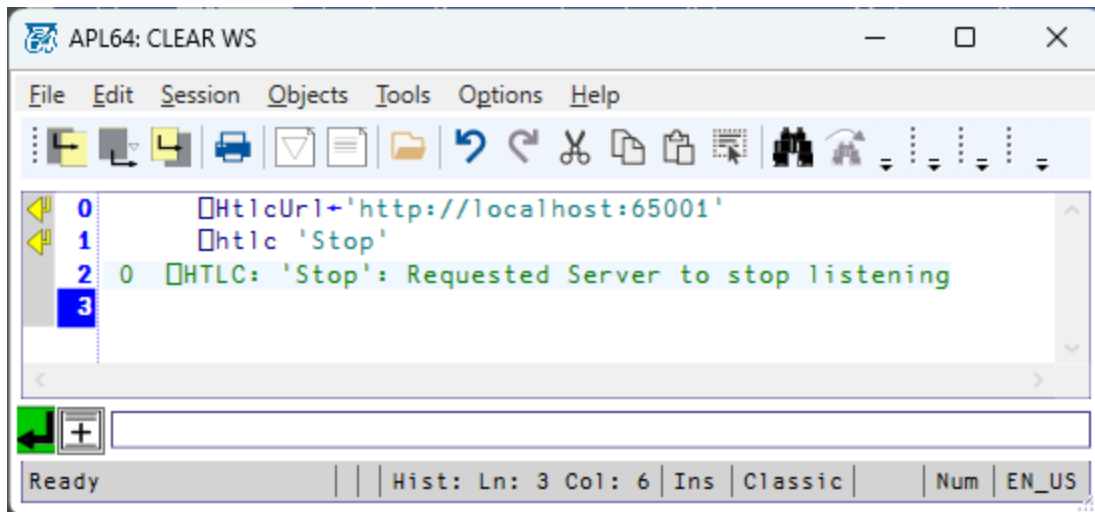


On another instance of the APL64 Developer version, create an HTLC client and try to stop the public server. The client displays the message that the request to stop the server has been made. The client does not wait for a message from the server, because if the server actually stopped, it could not send such a message to the client, and the client would need to wait until the expiration of the timeout to realize it failed.

```

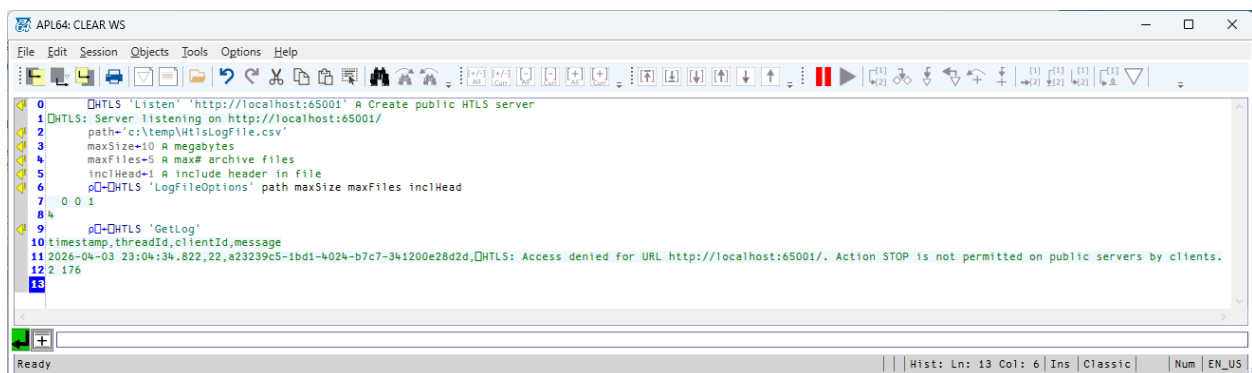
⊞ HtlcUrl←'http://localhost:65001'
⊞ htlc 'Stop'

```



Because a client of a public server cannot stop that server, the action will fail. The server logs the message that the stop action failed on the server side. The server can obtain the current log file content at any time. On the HTLS server, obtain the current log file content:

$\rho \leftarrow \text{HTLS 'GetLog'}$



Meanwhile the client wants to check if the server actually stopped, so the client gets the portion of the HTLS server log which is associated with the HTLC client indicating the server did not stop.

$\rho \leftarrow \text{HTLC 'GetServerLog'}$

```

APL64: CLEAR WS
File Edit Session Objects Tools Options Help
0 [HTLS 'http://localhost:65001/'
1 [HTLS 'Stop'
2 0 [HTLS: 'Stop': Requested Server to stop listening
3 [HTLS 'GetServerLog'
4 0 timestamp,threadId,clientId,message
5 2026-04-03 23:04:34.822,22,a23239c5-1bd1-4024-b7c7-341200e28d2d,[HTLS: Access denied for URL http://localhost:65001/. Action STOP is not permitted on public servers by clients.
6
Ready | Hist: Ln: 6 Col: 5 | Ins | Classic | Num | EN_US

```

Help (?)

The Help action returns the summary documentation of the `[HTLS` system function.

Syntax: `charMat ← [HTLS 'Help'`

`charMat ← [HTLS '?'`

`[HTLS '?'`

```

APL64: CLEAR WS
File Edit Session Objects Tools Options Help
0 [HTLS '?'
1 [HTLS actions:
2
3 charMat      + [HTLS 'Help'
4 charMat      + [HTLS '?'
5 msg          + [HTLS 'Listen' serverUrl
6 msg          + [HTLS 'Listen' (url1 url2 ... urlN)
7 msg          + [HTLS 'Listen' serverHttpsUrl certificatePath certificatePassword
8 msg          + [HTLS 'Listen' (url1 url2 ... urlN) certificatePath certificatePassword
9 msg          + [HTLS 'Pause'
10 msg          + [HTLS 'Pause' url1 [url2 ...]
11 msg          + [HTLS 'Resume'
12 msg          + [HTLS 'Resume' url1 [url2 ...]
13 (configUrls listenUrls) + [HTLS 'UrlInfo'
14 msg          + [HTLS 'ServerOff'
15 msg          + [HTLS 'Stop'
16 logLines(csv) + [HTLS 'GetLog'
17 msg          + [HTLS 'ClearLog'
18 (p sz n h)    + [HTLS 'LogFileOptions' [path maxSize maxFiles includeHeader]
19
20 serverUrl: http://serverName:port# or https://serverName:port#
21 (url1 url2 ... urlN): array of server URLs for multiple endpoints
22 certificatePath: path to .pfx or .p12 certificate file for HTTPS
23 certificatePassword: password for the certificate file
24 msg: char[] indicating success or failure
25 logLines: character matrix of log entries (CSV format)
26 path: file path to the log file (empty string disables logging)
27 maxSize: maximum log file size in megabytes before rotation (0=no rotation)
28 maxFiles: maximum archived log files to keep (0=unlimited)
29 includeHeader: 1 to include CSV header, 0 to omit
30
31 Note: Logging is disabled by default. Use 'LogFileOptions' to enable.
32 Note: Archives are named {basename}_{yyyyMMdd_HHMMss}.csv
33
34 Examples:
35 [HTLS 'Listen' 'http://localhost:8080/'
36 [HTLS 'Listen' ('http://localhost:8080/' 'http://192.168.1.100:8080/')
37 csv←[HTLS 'GetLog' A Get current log content
38 (p s f h)←[HTLS 'LogFileOptions' A Get current settings
39 [HTLS 'LogFileOptions' ' ' 0 0 0 A Disable logging
40 [HTLS 'LogFileOptions' 'C:\MyLogs\htls.csv' 10 5 1 A 10MB, 5 archives, with header
41
Ready | Hist: Ln: 41 Col: 6 | Ins | Classic | Cap | Num | EN_US

```

Listen

The Listen action requests the ☐HTLS server to listen to incoming client requests on the specified URL(s).

Single url:

```
msg ← ☐HTLS 'Listen' serverUrl [certificatePath certificatePassword]
```

Multiple urls:

```
msg ← ☐HTLS 'Listen' (url1 url2 ... urlN) [certificatePath certificatePassword]
```

The server instance of APL64 will be listening to incoming client requests on the specified serverUrl(s).

Urls are of the form: http://serverName:port#. The serverName can be a name or an IP address of the server machine, e.g. http://myserver:65001 or <http://192.168.1.107:65001>.

The port number, if any, should be selected so that it does not interfere with other http traffic reaching the server machine.

certificatePath and certificatePassword are optional arguments used when the https protocol is needed. The certificatePath can point to a .pfx- or .p12-format certificate file. A [self-signed certificate](#) can be used for development purposes. The certificate specified by the certificate path must be suitable for all the https urls on which the server is listening.

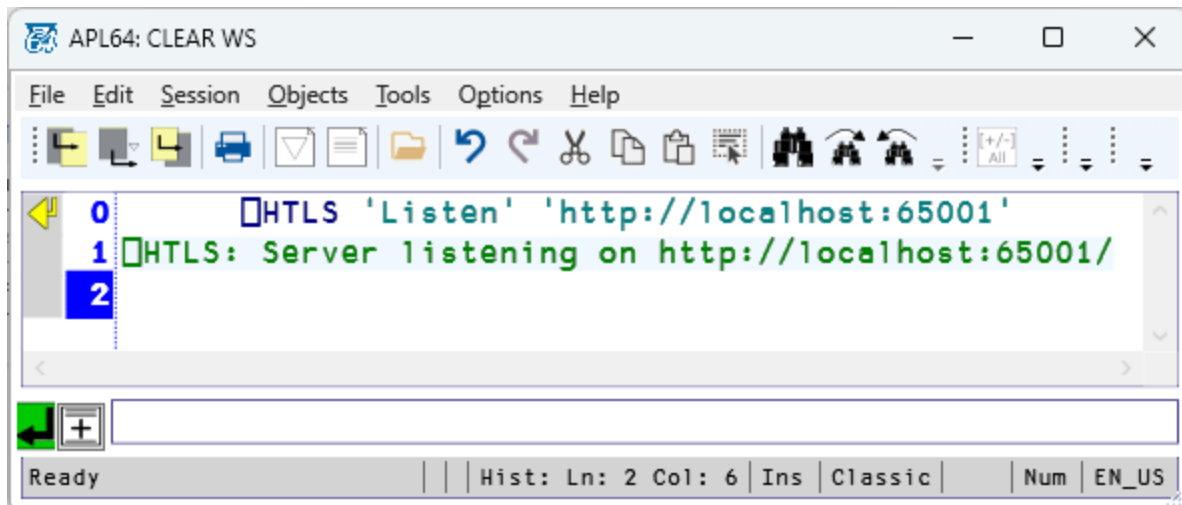
A server created using the Listen action is a public server which can be accessed by any ☐HTLC client.

Subsequent use of the ☐HTLS 'Listen' action for the same APL64 instance, will stop the server from listening to the previous url(s) and start listening on the new url(s).

If the server is not on the same workstation or local network as the client, it is a remote server. For a remote server permissions, security and port forwarding need to be considered so that client requests can reach the server and server responses can reach the client.

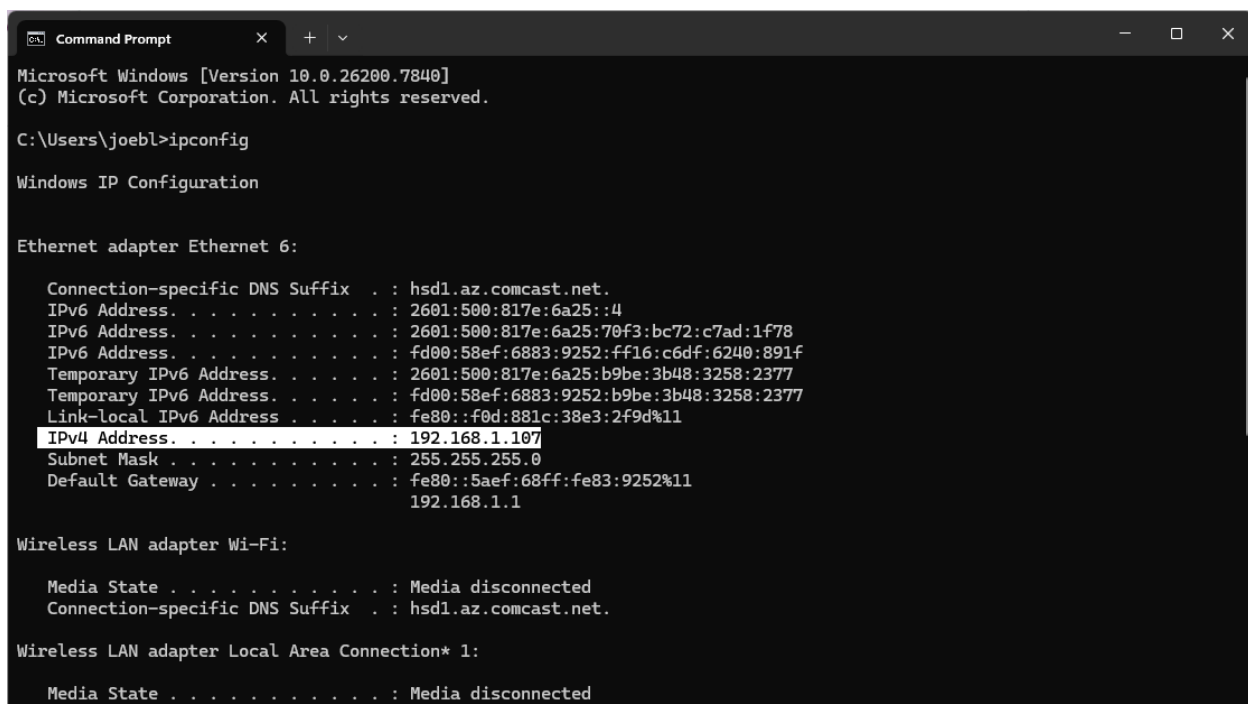
Example #1: Listen on localhost server

```
☐HTLS 'Listen' 'http://localhost:65001'
```



Example #2: Listen on a named server, not localhost server

Find the IPv4 address of the server machine:

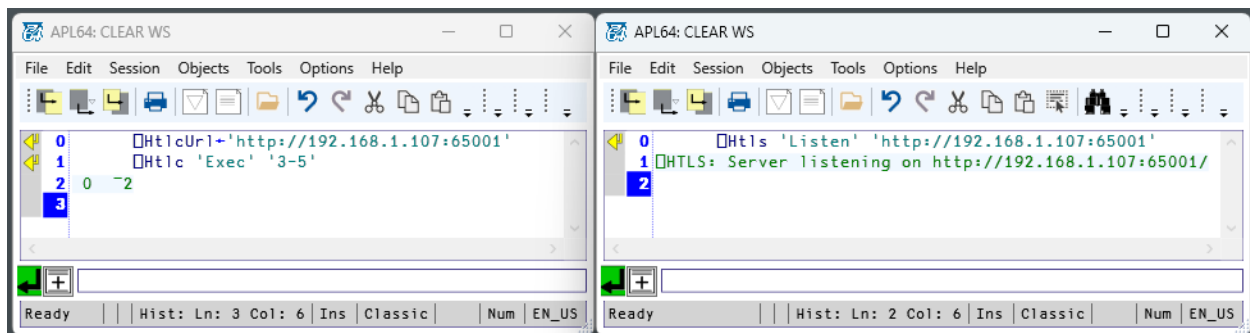


Create the server and start listening on the url:

```
☐ Htls 'Listen' 'http://192.168.1.107:65001'
```

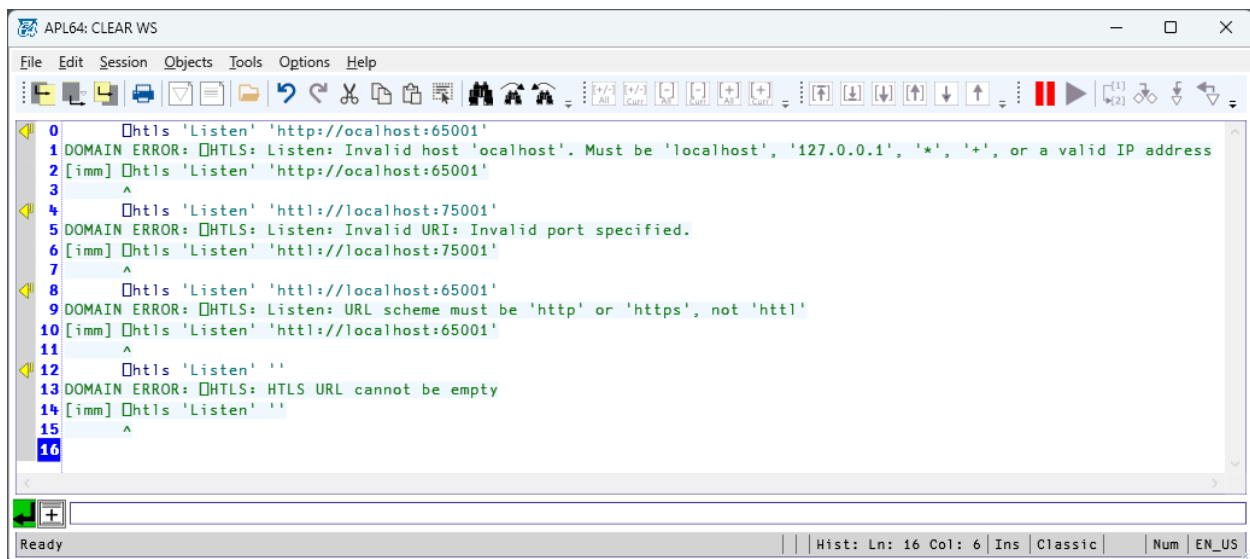
On the client-side access the server to do work:

```
☐ HtlcUrl<'http://192.168.1.107:65001'
☐ Htlc 'Exec' '3-5'
```



Example #3: Exception handling by the server-side 'Listen' action

- ☐ htls 'Listen' 'http://localhost:65001'
- ☐ htls 'Listen' 'http://localhost:75001'
- ☐ htls 'Listen' 'http://localhost:65001'
- ☐ htls 'Listen' ''



Example #4: Multiple Servers

In this example the ☐Htlc 'ListenLocal' action is used to create multiple independent servers. Another APL64 instance is created and the ☐Htlc 'Listen' is used to make it an additional server. Each server can have a different workspace loaded and perform different actions.

On the client side:

The client creates two local servers, each listening on separate urls.

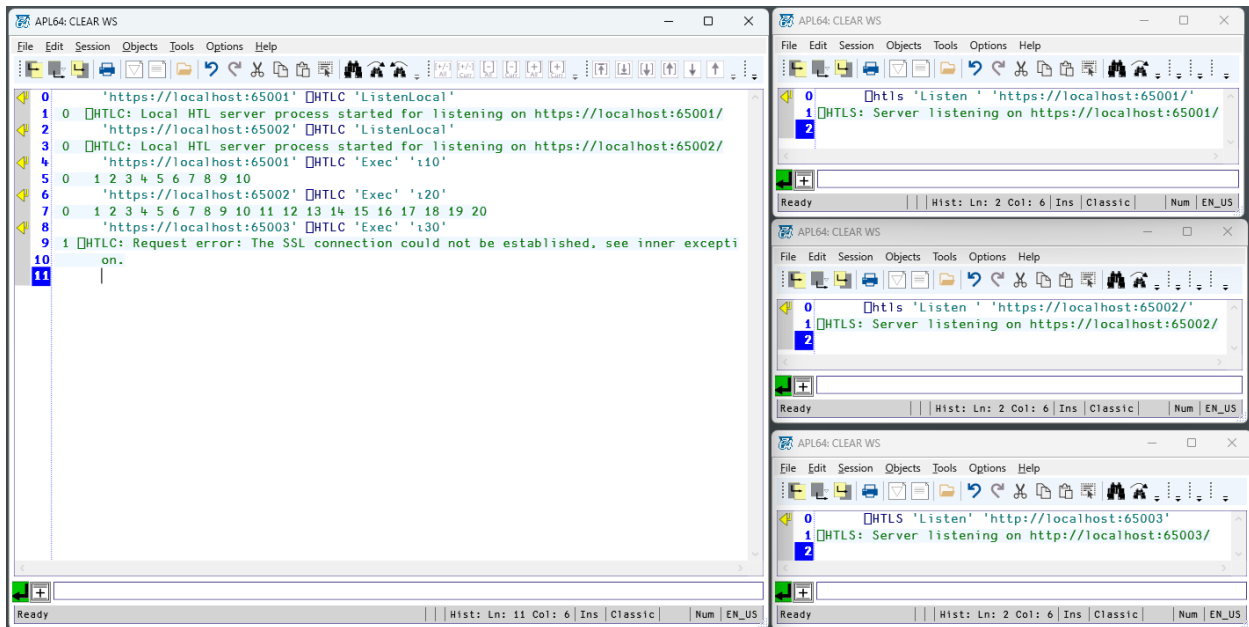
```
'http://localhost:65001' □ HTLC 'ListenLocal'
'http://localhost:65002' □ HTLC 'ListenLocal'
'http://localhost:65001' □ HTLC 'Exec' 'i10'
'http://localhost:65002' □ HTLC 'Exec' 'i20'
```

On the server side:

A separate APL64 instance starts listening on a third url.

```
□ HTLS 'Listen' 'http://localhost:65003'
```

There a total of four APL64 instances running in this example. The servers created by □ HTLC ListenLocal are exclusive servers for the client. The server created by the □ HTLS Listen action is a public server.



Example #5: Server listening on multiple urls

Establish server:

```
□ HTLS Server
□ htls 'Listen' ('http://localhost:65001' «http://localhost:65002»)
```

Establish two clients and make requests to the server:

```
□ HTLC Client #1
'http://localhost:65001' □ HTLC 'Exec' 'i10'
```

```
'http://localhost:65002' □ HTLC 'Exec' '10-10'
```

□ HTLC Client #2

```
'http://localhost:65001' □ HTLC 'Exec' '20-10'
```

```
'http://localhost:65002' □ HTLC 'Exec' '30-10'
```

The screenshot displays four windows of the APL64: CLEAR WS environment. The top-left window shows the code for 'HTLC Client #1' with three lines of code. The bottom-left window shows the code for 'HTLC Client #2' with three lines of code. The top-right window shows the code for 'HTLS Server' with two lines of code. The bottom-right window shows the execution output of the HTLS server, displaying a series of numbers and the text '[[HTLS: EXEC:ExecRes]'.

LogFileOptions

The LogFileOptions action sets the log file options for the □ HTLS server.

Syntax:

```
(p s f h) ← □ HTLS 'LogFileOptions'
```

Returns the log file options in effect

```
(p s f h) ← □ HTLS 'LogFileOptions' path maxSize maxFiles includeHeader
```

path: Path for current log file, or empty character vector for no logging.

maxSize: Maximum number of megabytes in the current log file

maxFiles: Maximum number of log file archives

includeHeader: 0/No 1/Yes

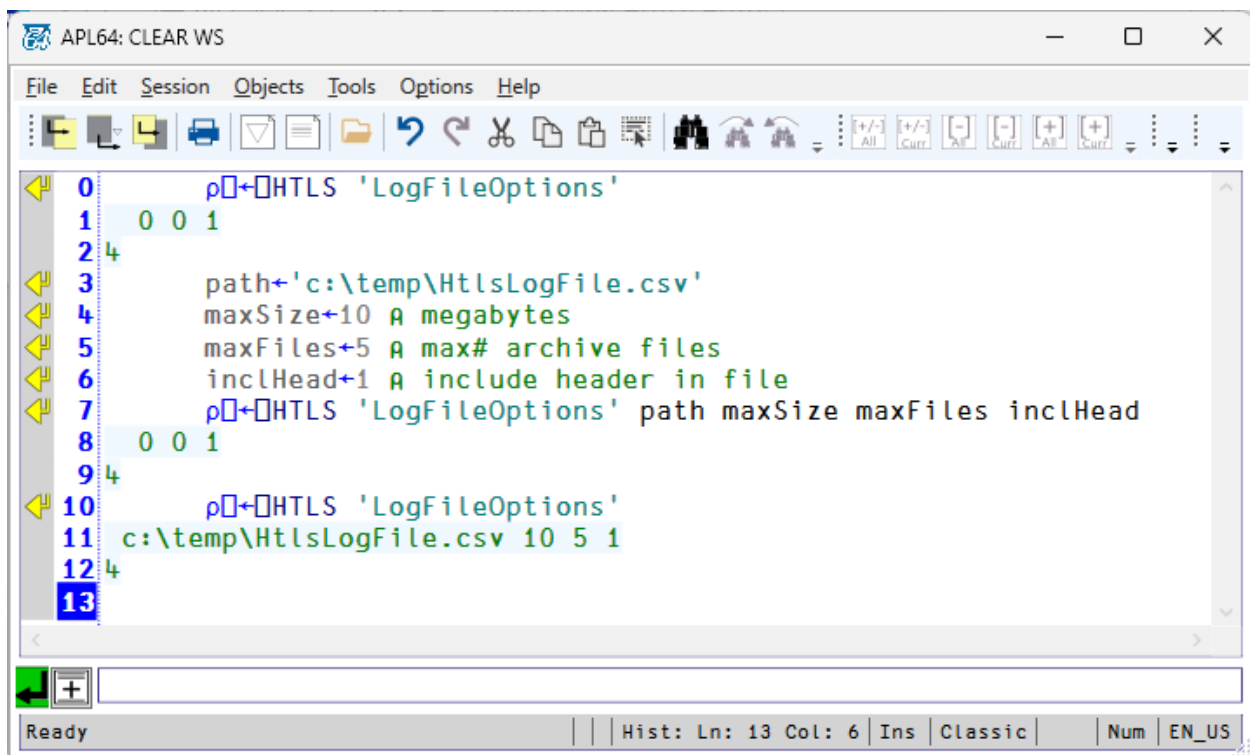
Servers are generally not continuously monitored by a person, so instead HTLS server-side exceptions can be logged for subsequent review.

If logging is enabled, by providing a log file path, the log file contains exceptions which occur on the HTLS server. If the maxSize is reached, the current log file is archived in the same location as the current log file, and a new current log file is created. If the maxFiles is reached, the oldest log file archives are deleted as necessary.

APL64 interpreter exceptions for HTLC client actions such as Exec or Call, are not logged on the server-side. They are returned to the HTLC client in the action result.

Example:

```
ρ⌈←⌈HTLS 'LogFileOptions'  
path←'c:\temp\HtIsLogFile.csv'  
maxSize←10 ⌈ megabytes  
maxFiles←5 ⌈ max# archive files  
inclHead←1 ⌈ include header in file  
ρ⌈←⌈HTLS 'LogFileOptions' path maxSize maxFiles inclHead  
ρ⌈←⌈HTLS 'LogFileOptions'
```



```
APL64: CLEAR WS  
File Edit Session Objects Tools Options Help  
0 ρ⌈←⌈HTLS 'LogFileOptions'  
1 0 0 1  
2 4  
3 path←'c:\temp\HtIsLogFile.csv'  
4 maxSize←10 ⌈ megabytes  
5 maxFiles←5 ⌈ max# archive files  
6 inclHead←1 ⌈ include header in file  
7 ρ⌈←⌈HTLS 'LogFileOptions' path maxSize maxFiles inclHead  
8 0 0 1  
9 4  
10 ρ⌈←⌈HTLS 'LogFileOptions'  
11 c:\temp\HtIsLogFile.csv 10 5 1  
12 4  
13  
Ready | Hist: Ln: 13 Col: 6 Ins | Classic | Num | EN_US
```

Messages logged by an ☐ HTLS server include:

- ☐ HTLS: Listen: Unexpected exception stopping old host: ...
- ☐ HTLS: Pause: Unexpected exception stopping host: ...
- ☐ HTLS: Resume: Unexpected exception stopping host:
- ☐ HTLS: Stop: Unexpected exception: ...
- ☐ HTLS: Error processing request: ...
- ☐ HTLS: Access denied for URL Requesting client ID ... does not match assigned client ID ...

- ☐ HTLS: Access denied. URL ... is a private local server owned by another client.
- ☐ HTLS: Access denied for URL ... Action ... is not permitted on public servers by clients.
- ☐ HTLS: Server listening stopped by client id: ...
- ☐ HTLS: Server initiating ServerOff action requested by client id: ...
- ☐ HTLS: Unexpected exception attempting client-requested action

Pause

The Pause action pauses the listening of the ☐ HTLS server on all or programmer-specified url(s).

msg ← ☐ HTLS 'Pause'

msg ← ☐ HTLS 'Pause' url1 [url2 ...]

Note: ☐ HTLC clients of the server will receive exceptions while the server is paused.

Example: See the Resume action example

Resume

The Resume action resumes the listening of the ☐ HTLS server on all or programmer-specified url(s) which were previously listening.

msg ← ☐ HTLS 'Resume'

msg ← ☐ HTLS 'Resume' url1 [url2 ...]

Note: Once the server is resumed, ☐ HTLC clients of the server can utilize the server.

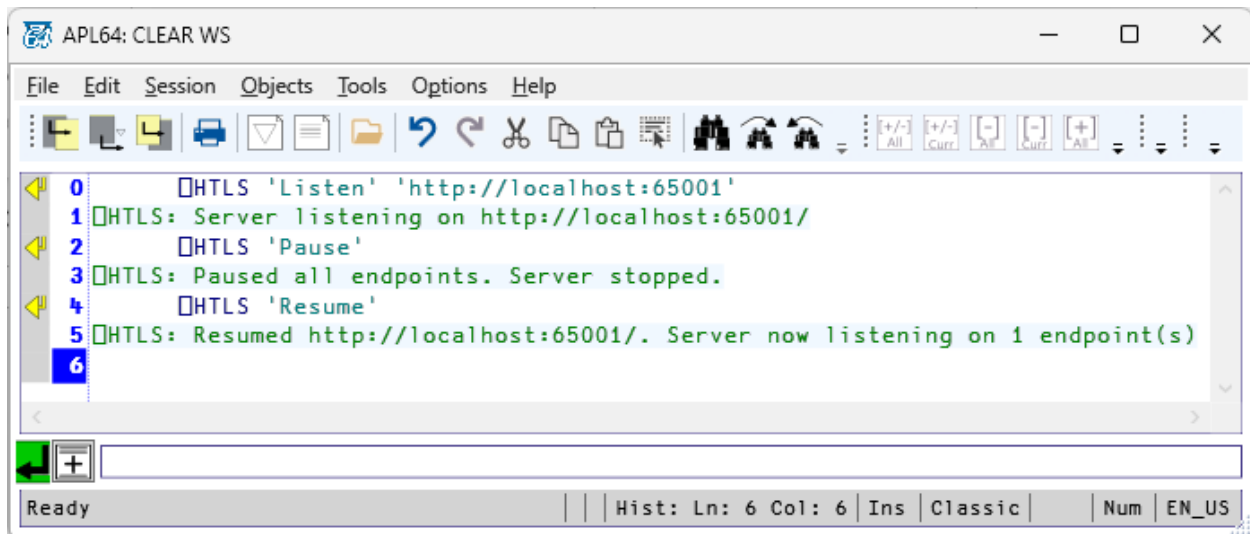
Example: Pause / Resume all urls

Server: Execute these statements one-at-a-time:

☐ HTLS 'Listen' 'http://localhost:65001'

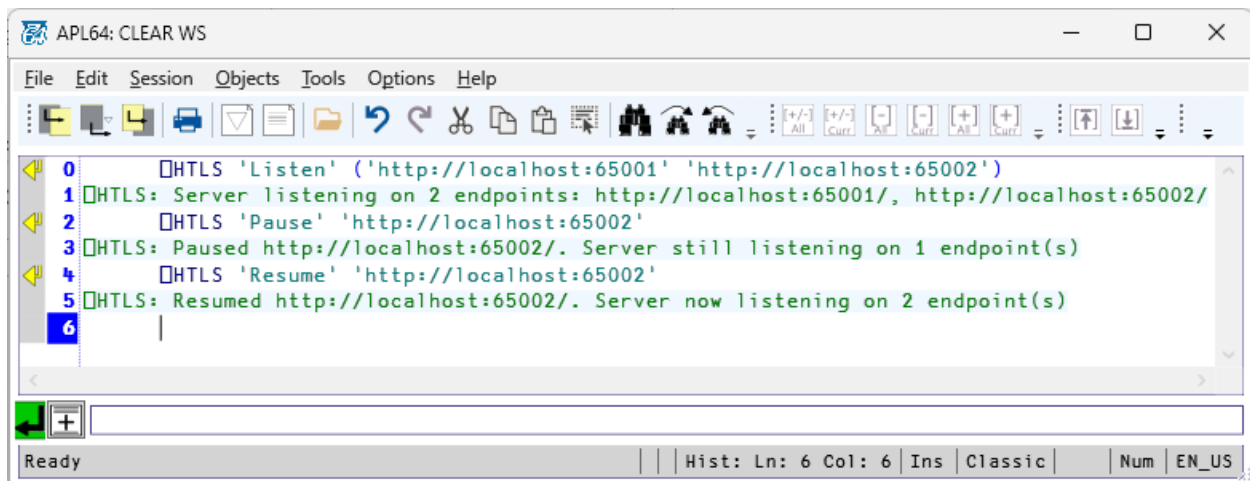
☐ HTLS 'Pause'

☐ HTLS 'Resume'



Example: Pause / Resume selected url(s)

- ☐ HTLS 'Listen' ('http://localhost:65001' 'http://localhost:65002')
- ☐ HTLS 'Pause' 'http://localhost:65002'
- ☐ HTLS 'Resume' 'http://localhost:65002'



ServerOff

The ServerOff action requests the ☐ HTLS server to stop all listening to incoming client requests on the specified URL(s) and exit.

Syntax: (hasErr errMsg result) ← ☐ HTLS 'ServerOff'

Note: This action may take a few seconds to complete on the ☐ HTLS server.

- ☐ HTLS 'Listen' 'http://localhost:65001'
- ☐ HTLS 'ServerOff' Click Enter to end server's APL instance

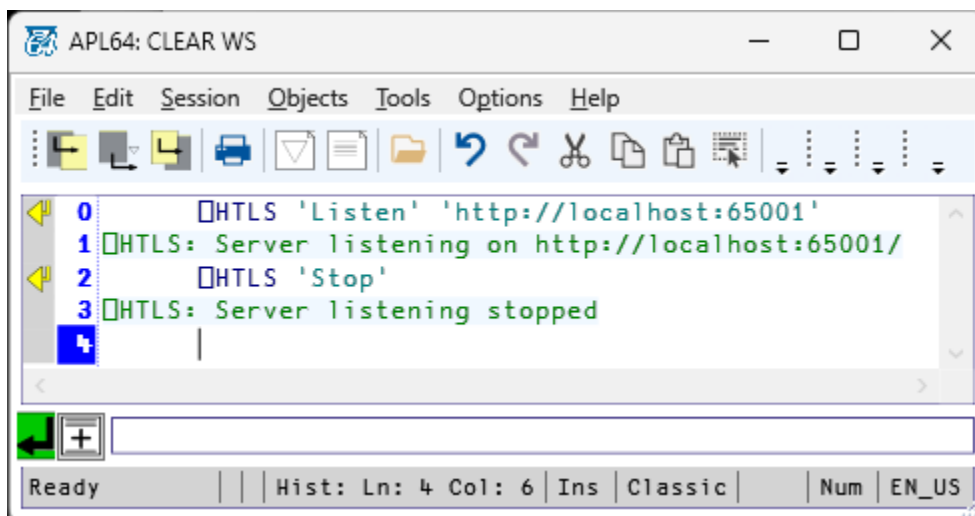
Stop

The Stop action requests the ☐HTLS server to stop listening to incoming client requests on the specified URL(s).

Syntax: msg ← ☐HTLS 'Stop'

Note: The server instance of APL64 is not ended.

```
☐HTLS 'Listen' 'http://localhost:65001'  
☐HTLS 'Stop'
```



UrlInfo

The UrlInfo action returns the currently configured and listened to urls.

Syntax: (configUrls listenUrls) ← ☐HTLS 'UrlInfo'

The UrlInfo action returns a two-element vector. The first vector element is a character array of the currently configured url(s), including the listening and paused url(s). The second vector element is a character array of the currently listened to urls. Because an ☐HTLS server is an asynchronous, multi-threaded server, the UrlInfo action is 'snap-shot' information which can change at any time depending on the actions performed upon the server.

Example:

```
☐HTLS 'Listen' ('http://localhost:65001' 'http://localhost:65002')  
☐HTLS 'UrlInfo'
```

```

□ HTLS 'Pause' 'http://localhost:65002'
□ HTLS 'UrlInfo'
□ HTLS 'Pause'
□ HTLS 'UrlInfo'
□ HTLS 'Resume' 'http://localhost:65002'
□ HTLS 'UrlInfo'
□ HTLS 'Resume'
□ HTLS 'UrlInfo'

```

```

APL64: CLEAR WS
File Edit Session Objects Tools Options Help
□ HTLS 'Listen' ('http://localhost:65001' 'http://localhost:65002')
1 □ HTLS: Server listening on 2 endpoints: http://localhost:65001/, http://localhost:65002/
2 □ HTLS 'UrlInfo'
3 http://localhost:65001/ http://localhost:65001/
4 http://localhost:65002/ http://localhost:65002/
5 □ HTLS 'Pause' 'http://localhost:65002'
6 □ HTLS: Paused http://localhost:65002/. Server still listening on 1 endpoint(s)
7 □ HTLS 'UrlInfo'
8 http://localhost:65001/ http://localhost:65001/
9 http://localhost:65002/
10 □ HTLS 'Pause'
11 □ HTLS: Paused all endpoints. Server stopped.
12 □ HTLS 'UrlInfo'
13 http://localhost:65001/
14 http://localhost:65002/
15 □ HTLS 'Resume' 'http://localhost:65002'
16 □ HTLS: Resumed http://localhost:65002/. Server now listening on 1 endpoint(s)
17 □ HTLS 'UrlInfo'
18 http://localhost:65001/ http://localhost:65002/
19 http://localhost:65002/
20 □ HTLS 'Resume'
21 □ HTLS: Resumed http://localhost:65001/. Server now listening on 2 endpoint(s) (http://localhost:65002/ already active)
22 □ HTLS 'UrlInfo'
23 http://localhost:65001/ http://localhost:65002/
24 http://localhost:65002/ http://localhost:65001/
25

```

□ HTLCURL

Many of the □ HTLC actions require a URL left argument. Instead of specifying this URL for each □ HTLC action, □ HTLCURL can be set to contain the required URL value. When □ HTLCURL has a value, no left argument URL value is needed for those □ HTLC actions which require a URL value.

Syntax:

□ HTLCURL ← URLText

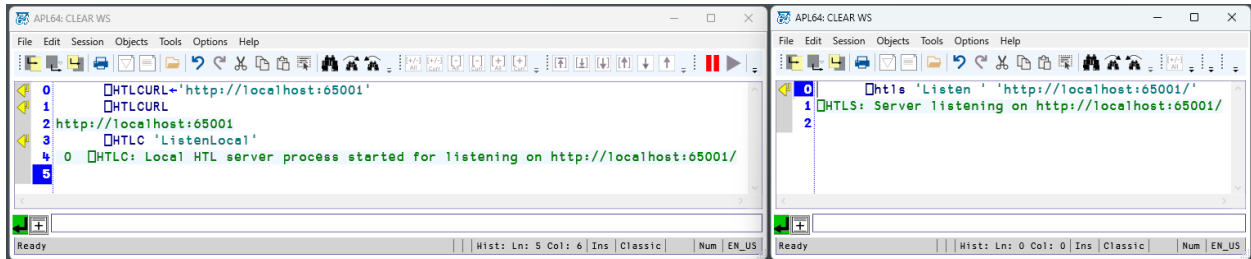
varName ← □ HTLCURL

Example:

```

□HTLCURL←'http://localhost:65001'
□HTLCURL
□HTLC 'ListenLocal'

```



Development and Runtime Use of □HTLC and □HTLS

Development Environment

The APL64 Developer version provides the APL64 development environment. All □HTLC and □HTLS system function actions can be used in the APL64 development environment. When used in the APL64 development environment, the □HTLS server can be made visible and debugged.

Runtime Environment

APL64 has two runtime environments designed for applications in production use. The APL64 production environments are available as a Windows Runtime Executable (WRE) or a Cross-platform Component (CPC).

- All □HTLC system function actions, except for ListenLocal, can be used in an APL64 WRE or CPC.
- All □HTLS system function actions, including Listen, can be used in an APL64 WRE or CPC.

Example #1: Using □HTLS in a WRE

In this example an APL64 WRE (Windows Runtime Executable) is used to start an APL64 □HTLS server.

Create the WRE (C:\HtIsWreTest\Source\HtIsWreTest.ws64) workspace

Create the WRE workspace containing the required □LX function:

```

LXFN
HTLS_URL←'http://localhost:65001'
HTLS_FormName←'HTLS_FORM'

```

```

HTLS_FormName □wi 'Create' 'Form' ('size' (20 80))
HTLS_FormName □wi 'onClose' '□sa←"OFF"'
Ⓜ↑ Essential to end listening when this form is closed!
:TRY
  □HTLS 'Listen' HTLS_URL
  ←(HTLS_FormName, '.Label1') □wi 'Create' 'Label' ('caption' ('Listening on: ', HTLS_URL))
  ('where' (1 1 2 75))
:CATCH
  ←(HTLS_FormName, '.Label1') □wi 'Create' 'Label' ('caption' ('Url: ', HTLS_URL)) ('where' (1
  1 2 75))
  ←(HTLS_FormName, '.Edit') □wi 'Create' 'Edit' ('style' (+/4 16 64 8192)) ('text' ('Listening
  Exception: ', □EM)) ('where' (3 1 16 75))
:ENDTRY
'HTLS_FORM' □wi 'Wait'

```

A waiting □wi Form or a continuous-loop in the □LX function is needed to prevent the WRE from closing.

```

APL64: C:\HtIsWreTest\Source\HtIsWreTest.ws64
File Edit Session Objects Tools Options Help
V:LXFN
0 LXFN
1 HTLS_URL←http://localhost:65001
2 HTLS_FormName←HTLS_FORM
3 HTLS_FormName □wi 'Create' 'Form' ('size' (20 80))
4 HTLS_FormName □wi 'onClose' '□sa←"OFF"'
5 At Essential to end listening when this form is closed!
6 :TRY
7   □HTLS 'Listen' HTLS_URL
8   ←(HTLS_FormName, '.Label1') □wi 'Create' 'Label' ('caption' ('Listening on: ', HTLS_URL)) ('where' (1 1 2 75))
9 :CATCH
10  ←(HTLS_FormName, '.Label1') □wi 'Create' 'Label' ('caption' ('Url: ', HTLS_URL)) ('where' (1 1 2 75))
11  ←(HTLS_FormName, '.Edit') □wi 'Create' 'Edit' ('style' (+/4 16 64 8192)) ('text' ('Listening Exception: ', □EM)) ('where' (3 1 16 75))
12 :ENDTRY
13 'HTLS_FORM' □wi 'Wait'
[1;10] Commit Changes Commit & Close
0 )xload C:\HtIsWreTest\Source\HtIsWreTest.ws64
1 "C:\HtIsWreTest\Source\HtIsWreTest.ws64" LAST SAVED 2/25/2026 3:35:43 AM
2 )fns
3 LXFN
4 )ed LXFN
5
Ready Hist: Ln: 5 Col: 6 Ins Classic Num EN_US

```

Complete the APL64 WRE creation utility dialog

APL64: Create Windows Runtime Executable

Runtime Workspace Filename: * C:\HtIsWreTest\Source\HtIsWreTest.ws64

EXECUTABLE CONTENT

Runtime Windows Executable Target Folder: * C:\HtIsWreTest\Target Browse

Runtime Executable Name: * Same as Runtime Workspace Name

Digital signature command: Digitally sign output

WRE Output Type: * Single Exe File Including All Dependencies

APL64 xml-format Configuration File: Browse Include APL64 xml-format Configuration File

APLNow32 adf-format Configuration File: C:\Users\joebf\source\repos\APLNowLLC\APL64\NetCore\bin\Debug\net10.0\APLNow32.adf Browse Include APLNow32 adf-format Configuration File

APLNow32 ini-format Configuration File: Browse Include APLNow32 ini-format Configuration File

APLNow32 Manifest File: C:\Users\joebf\source\repos\APLNowLLC\APL64\NetCore\bin\Debug\net10.0\APLNow32.exe.Manifest Browse Include APLNow32 manifest File

Additional Files Required for the Application

Source Path Base Target Path Target Path Suffix Overwrite

Add One Required File Add Multiple Required Files Add Folder of Required Files Remove All Required Files

ASSEMBLY META-DATA

Properties.Details: File Description: HtIsWreTest

Properties.Details: Product Name: HtIsWreTest

Properties.Details: Copyright: * ©APLNext LLC

Properties.Details: File Version: * 1.0.0.11

Properties.Details: Version: Same as File Version

Assembly.Info: Company Name: APLNext LLC

Assembly.Info: Application Description: Same as File Description

Assembly.Info: Version: Same as File Version

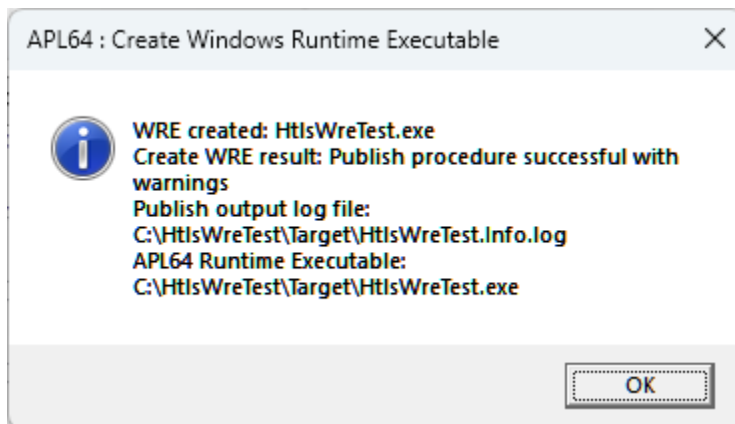
Assembly.Info: Neutral Language: EN-US

Assembly.Info: Description: Same as File Description

Application Icon File: Browse

Create Exit New WRE Info Load WRE Info Save WRE Info * Required entries

Click the WRE creation utility dialog 'Create' button

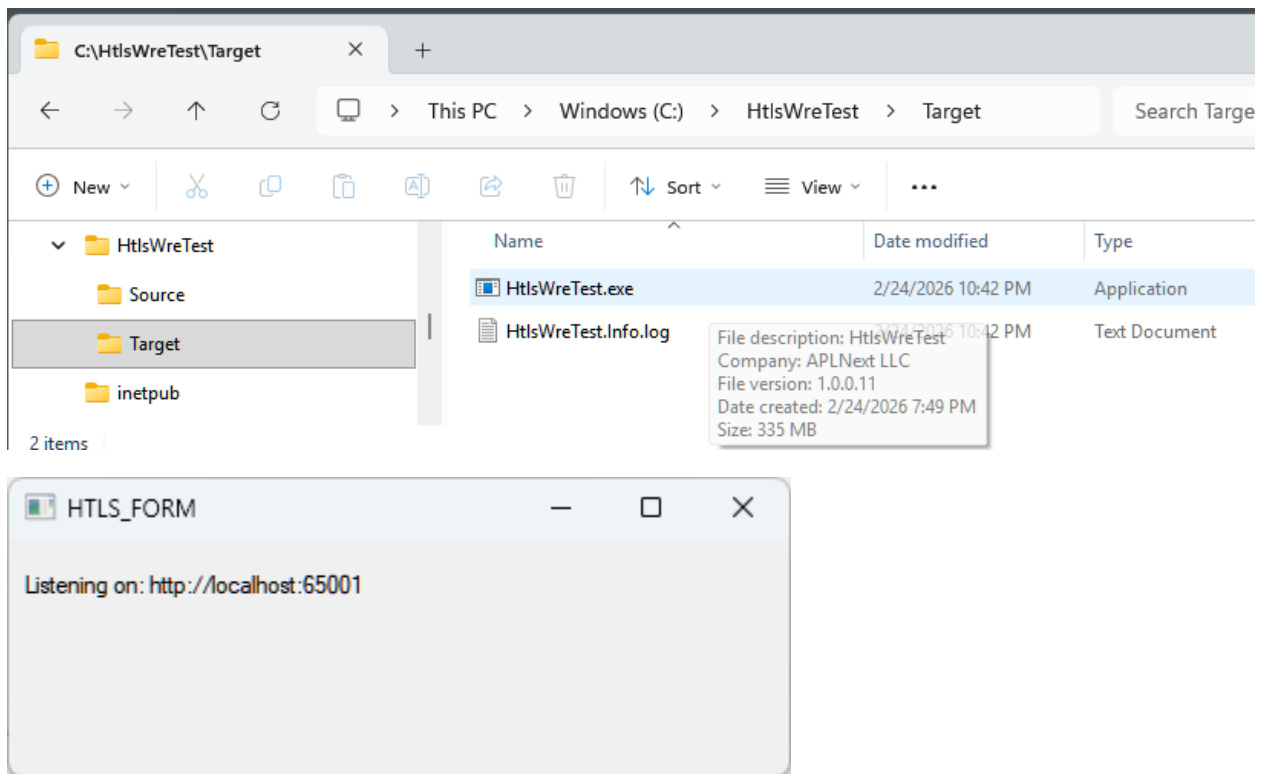


Save the WRE Creation Utility Input

Click the WRE creation utility 'Save WRE Info' button to save the WRE configuration information for future use.

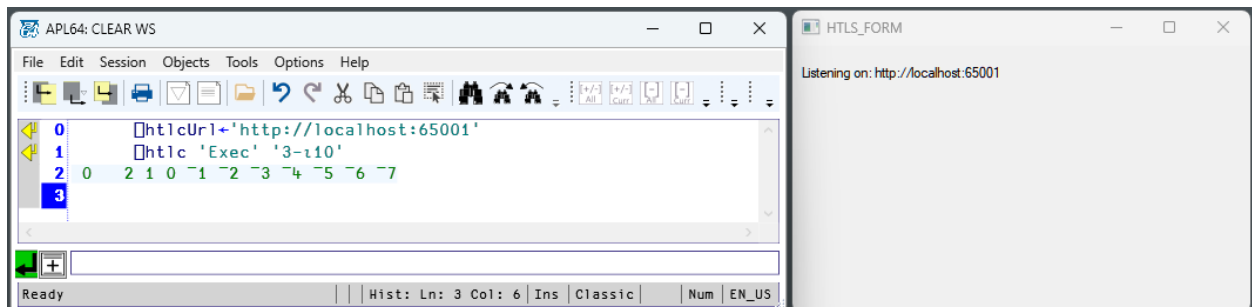
Run the APL64 WRE to create the ☐ HTLS Server

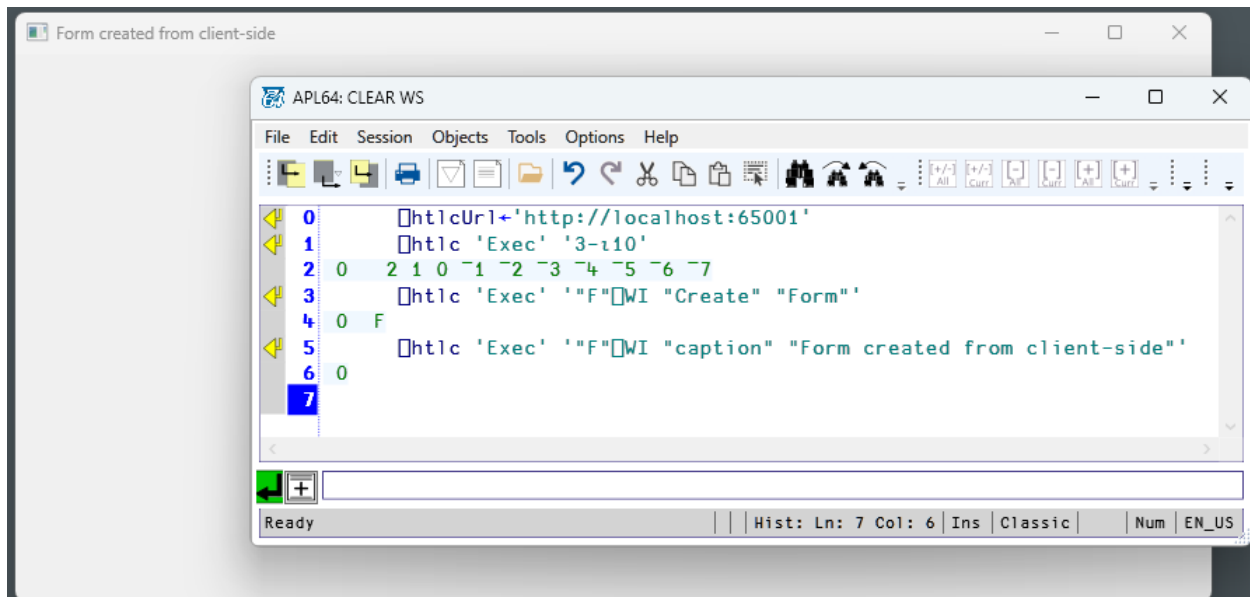
Double-click the Windows Runtime Executable file:



Use an APL64 Instance to access the WRE-based ☐HTLS Server

Create a separate APL64 instance and use ☐HTLC to access the public WRE-based ☐HTLS server:





The ☐HTLS server may be stopped by closing the ☐HTLS window.

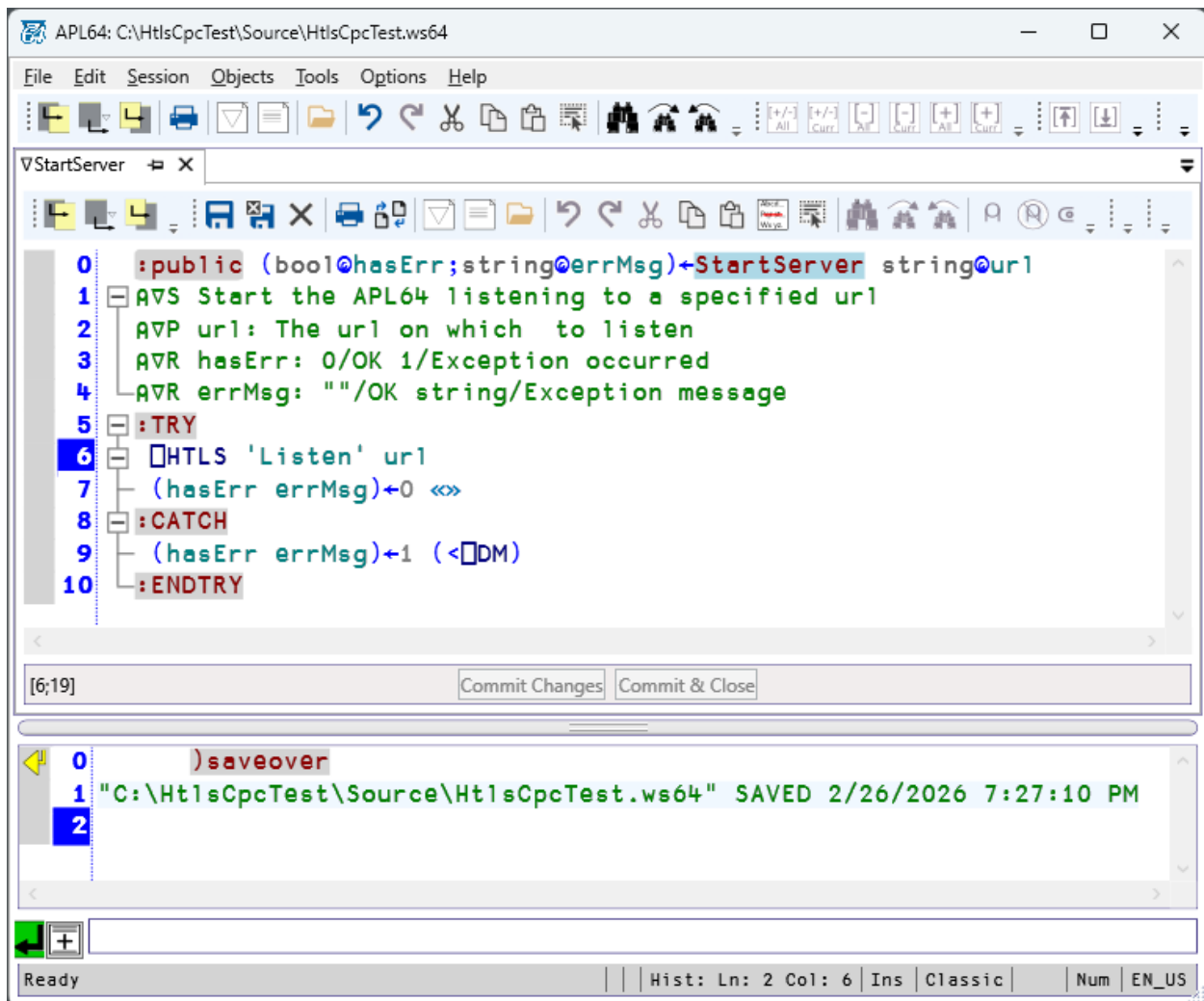
Example #2: ☐HTLS in a CPC

In this example an APL64 CPC is created to start an ☐HTLS server. This APL64 CPC can be used by a .Net project to start an ☐HTLS server.

Create the C:\HtIsCpcTest\Source\HtIsCpcTest.ws64 workspace

Create the required workspace with at least one public function, and include .Net Intellisense scaffold remarks:

```
:public (bool@hasErr;string@errMsg)←StartServer string@url
⌵VS Start the APL64 listening to a specified url
⌵VP url: The url on which to listen
⌵VR hasErr: 0/OK 1/Exception occurred
⌵VR errMsg: ""/OK string/Exception message
:TRY
  ☐HTLS 'Listen' url
  (hasErr errMsg)←0 «»
:CATCH
  (hasErr errMsg)←1 (<☐DM)
:ENDTRY
```



Complete the APL64 CPC creation dialog

APL64: Create Cross Platform Component

CPC Workspace Path *: C:\HtIsCpcTest\Source\HtIsCpcTest.ws64

CROSS PLATFORM COMPONENT CONTENT

Cross Platform Component Target Folder *: C:\HtIsCpcTest\Target Browse Open Folder in File Explorer

Cross Platform Component Name *: HtIsCpcTest ☒ Same as CPC Workspace Name

Cross Platform Component Class Name *: HtIsCpc Desired name of the CPC assembly without file path information and without file name extension. If this file exists in the CPC Target Folder it will be deleted at the start of the process. This value may be used as part of the CPC Nuget package id.

APL64 xml-format Configuration File: Browse ☐ Include APL64 xml-format Configuration File

Additional Files Required for the Application

Source Path Base Target Path Target Path Suffix Overwrite

Add One Required File Add Multiple Required Files Add Folder of Required Files Remove All Required Files

ASSEMBLY META-DATA

Properties:Details: File Description: HtIsCpcTest

Properties:Details: Product Name *: HtIsCpcTest

Properties:Details: Copyright *: ©APLNext LLC

Properties:Details: File Version *: 0.0.0.1

Properties:Details: Version: 0.0.0.1 ☒ Same as File Version

Assembly:Info: Company Name *: APLNext LLC

Assembly:Info: Application Description: HtIsCpcTest ☒ Same as File Description

Assembly:Info: Version: 0.0.0.1 ☒ Same as File Version

Assembly:Info: Neutral Language: EN-US

Assembly:Info: Description: HtIsCpcTest ☒ Same as File Description

Application Icon File: Browse

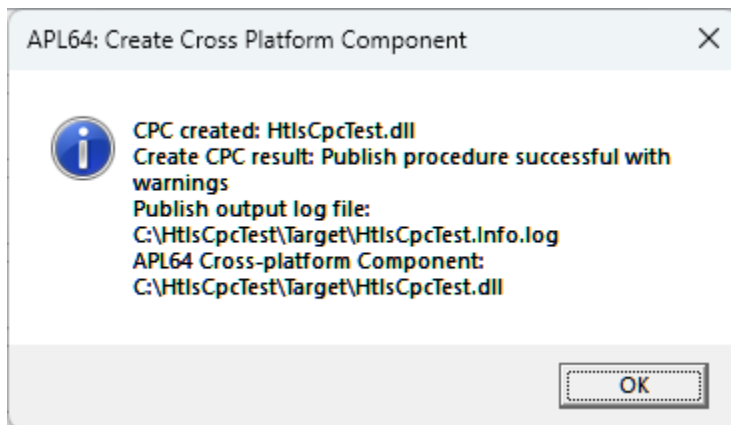
Nuget Package Guid *: e967b7ab-6c76-4aae-b103-40e83dada6f Create

Nuget Package Id*: APLNext_LLC.HtIsCpcTest ☒ Use CompanyName.ComponentName

Nuget Package Files

Create Exit New CPC Info Load CPC Info Save CPC Info * Required Entries

Click the CPC creation dialog 'Create' button

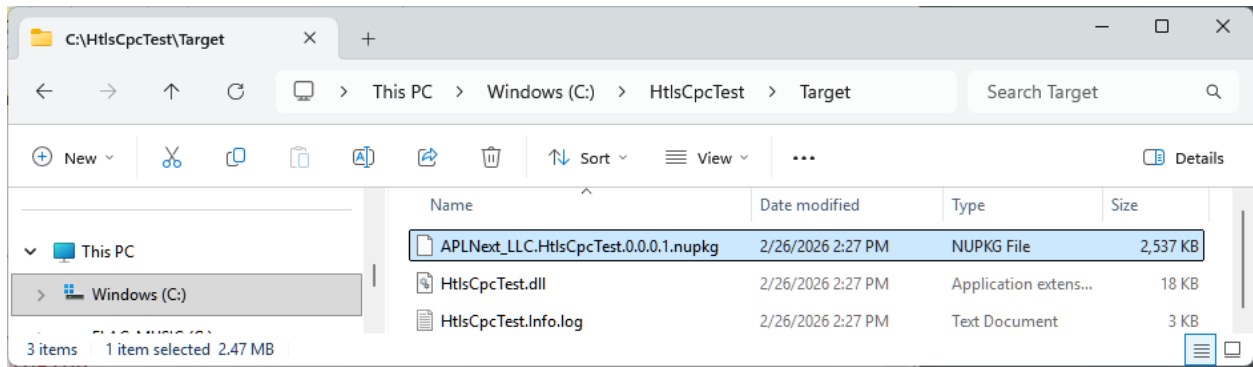


Save the CPC Creation Dialog Input

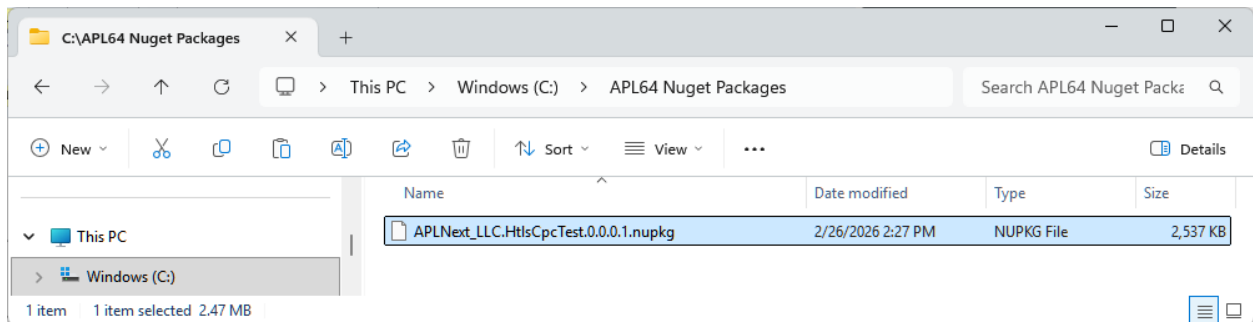
Click the CPC creation dialog 'Save CPC Info' button to preserve the CPC configuration for future use.

Copy the resulting Nuget package from the Target folder to the Nuget package publish folder:

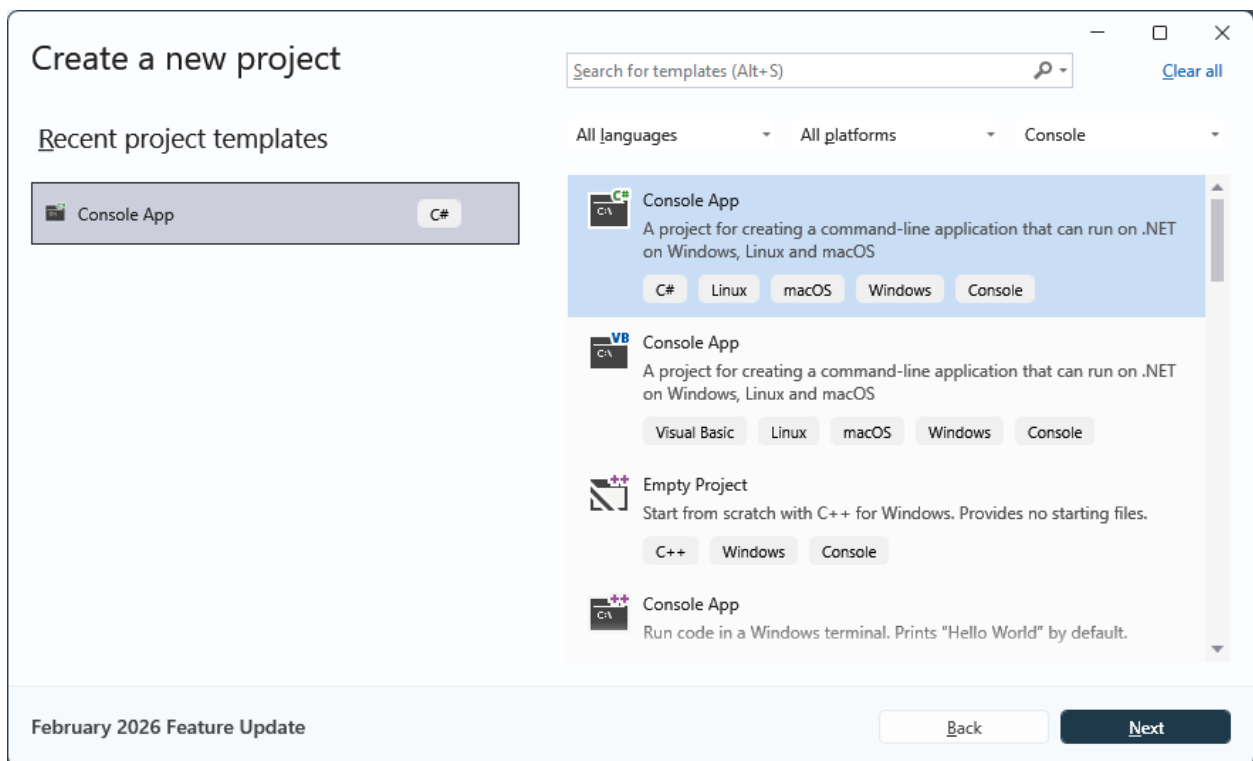
From:



To:



Use the APL64 CPC Nuget package in a .Net console project



Configure your new project

Console AppC#LinuxmacOSWindowsConsole

Project name

HttpCpcTestConsole

Location

C:\HttpCpcTest\HttpCpcTestConsole\

Solution name ⓘ

HttpCpcTestConsole

☐ Place solution and project in the same directory

Project will be created in "C:\HttpCpcTest\HttpCpcTestConsole\HttpCpcTestConsole\HttpCpcTestConsole\"

February 2026 Feature UpdateBackNext

Additional information

Console AppC#LinuxmacOSWindowsConsole

Framework ⓘ

.NET 10.0 (Long Term Support)

☐ Enable container support ⓘ

Container OS ⓘ

Linux

Container build type ⓘ

Dockerfile

☒ Do not use top-level statements ⓘ

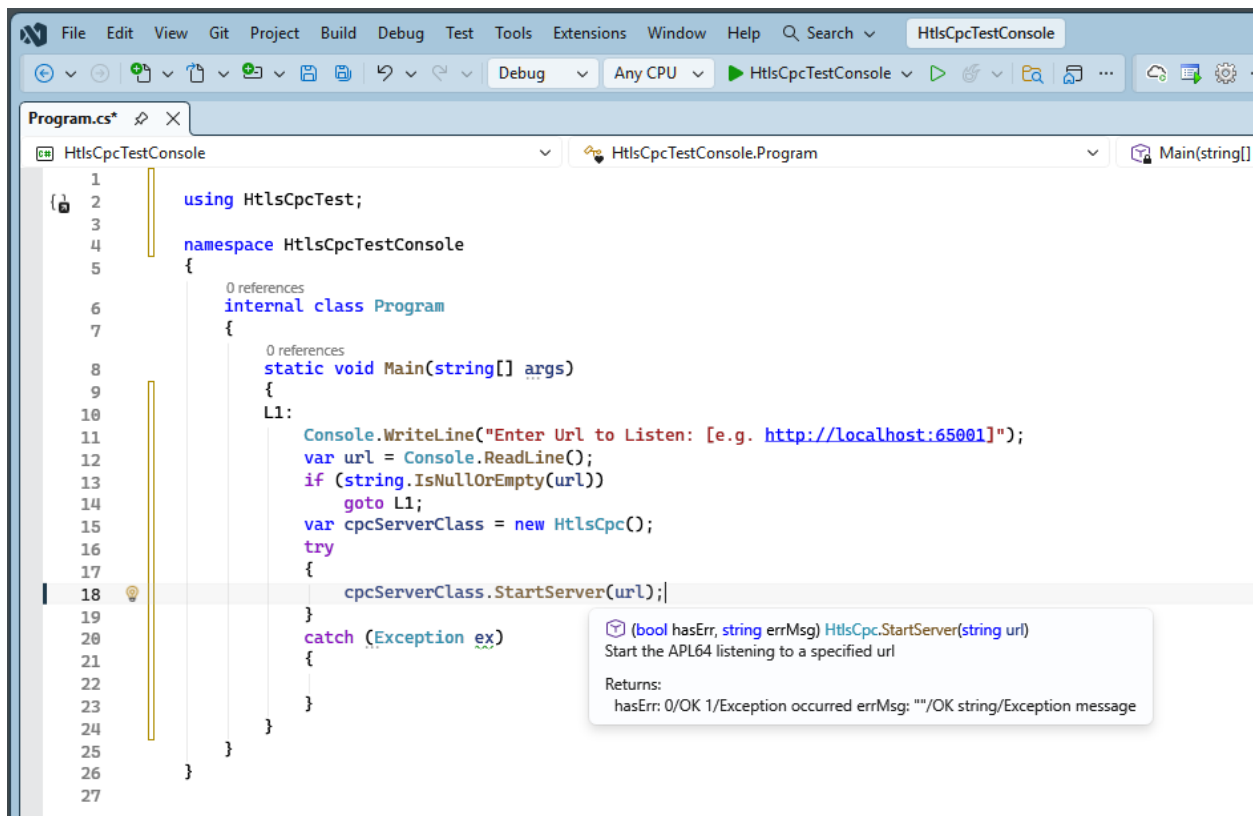
☐ Enable native AOT publish ⓘ

February 2026 Feature UpdateBackCreate

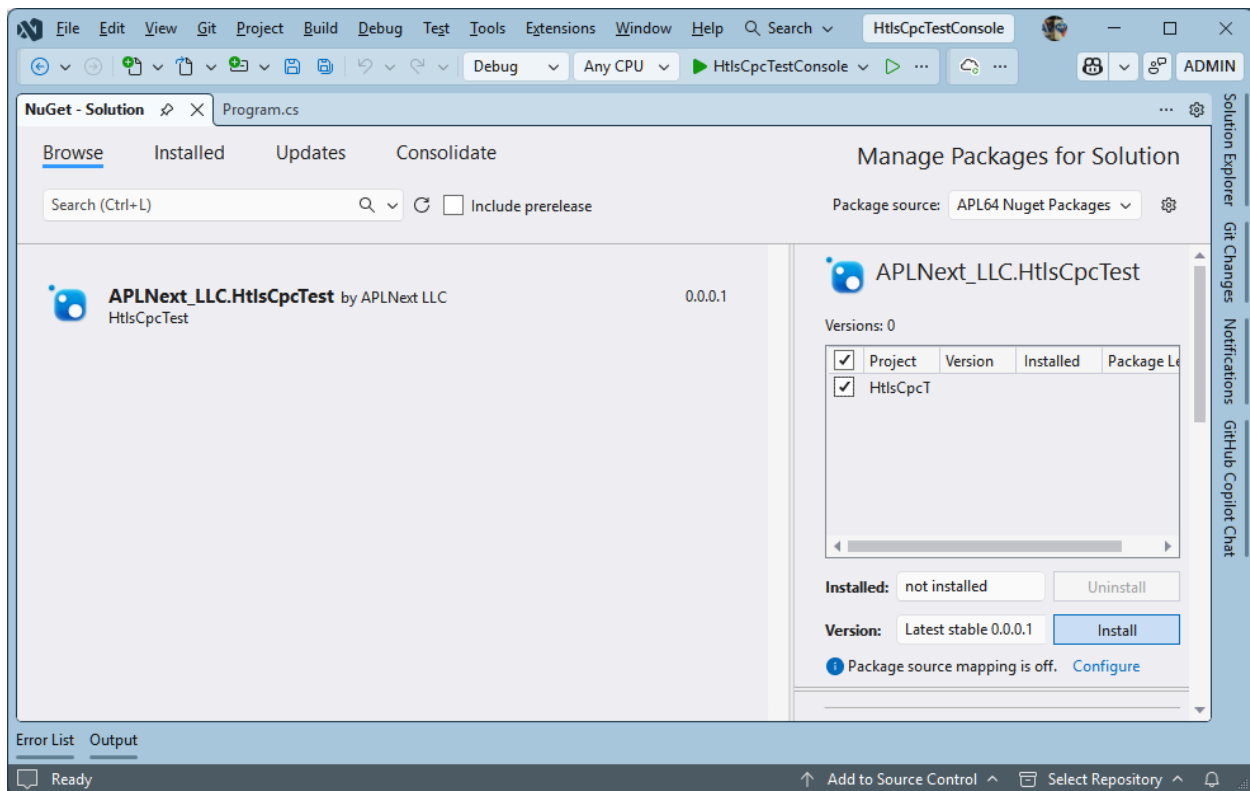
Replace the default C# source code with this code:

```
using HttpCpcTest;
namespace HttpCpcTestConsole
```

```
{
    internal class Program
    {
        static void Main(string[] args)
        {
            L1:
            Console.WriteLine("Enter Url to Listen: [e.g. http://localhost:65001]");
            var url = Console.ReadLine();
            if (string.IsNullOrEmpty(url))
                goto L1;
            var cpcServerClass = new HtlsCpc();
            try
            {
                cpcServerClass.StartServer(url);
            }
            catch (Exception ex)
            {
            }
        }
    }
}
```

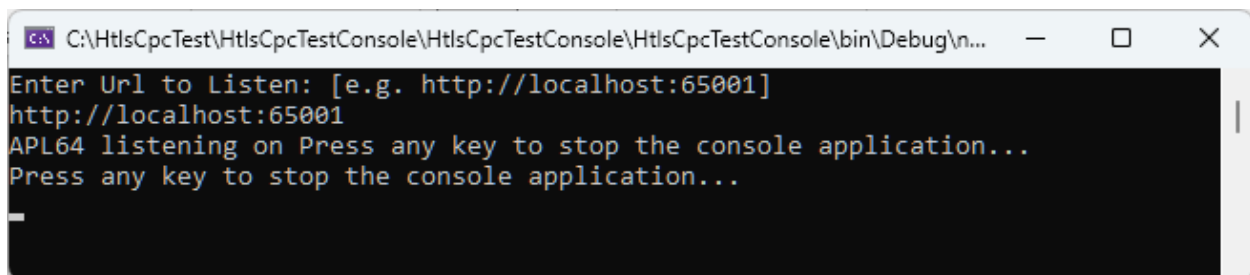


Use Tools > Nuget Package Manager > Manage Nuget Packages for Solution dialog to install the APL64 CPC Nuget package to the solution:



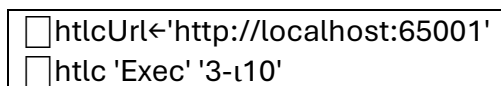
Run the console application using the APL64 CPC to Start the  HTLS Server

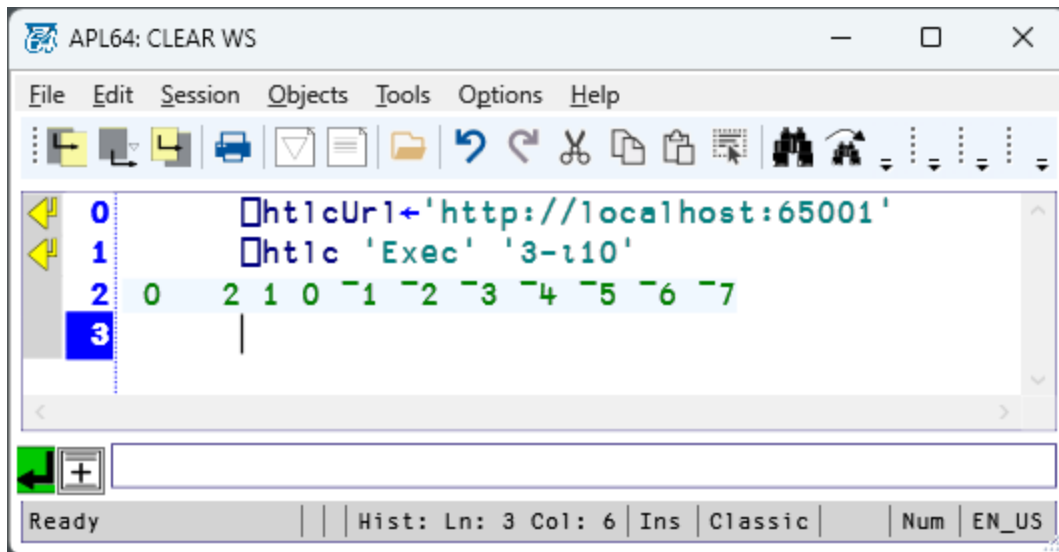
Run the console application and enter an appropriate url for listening:



Use an APL64 Developer Instance to Access the  HTLS Server

Start an APL64 instance and use  HTLC to access the CPC-based server:





Example #3: HTLC in a CPC

In this example APL64 public functions in an APL64 CPC are created for each applicable  HTLC action. The resulting APL64 CPC can be used to access an APL64  HTLS server from .Net using C#.

You can follow these instructions to build a copy of the CPC as a Nuget package which is a .Net client for an  HTLS server. The completed CPC Nuget package is also available here:

<https://aplnow.com/APL64/UserDocumentation/APLNext.HtlsNetClient.2026.6.0.4.zip>

Create the disk folders for the example

- C:\HTLS NET CLIENT
- C:\HTLS NET CLIENT\Source
- C:\HTLS NET CLIENT\Target

Create source workspace and public functions

Create the required workspace (C:\HTLS NET CLIENT\Source\HtlsNetClient.ws64) with these APL64 public functions, including .Net Intellisense scaffold remarks. Depending on your requirements, not all of these functions need to be created.

CallDyadic

```

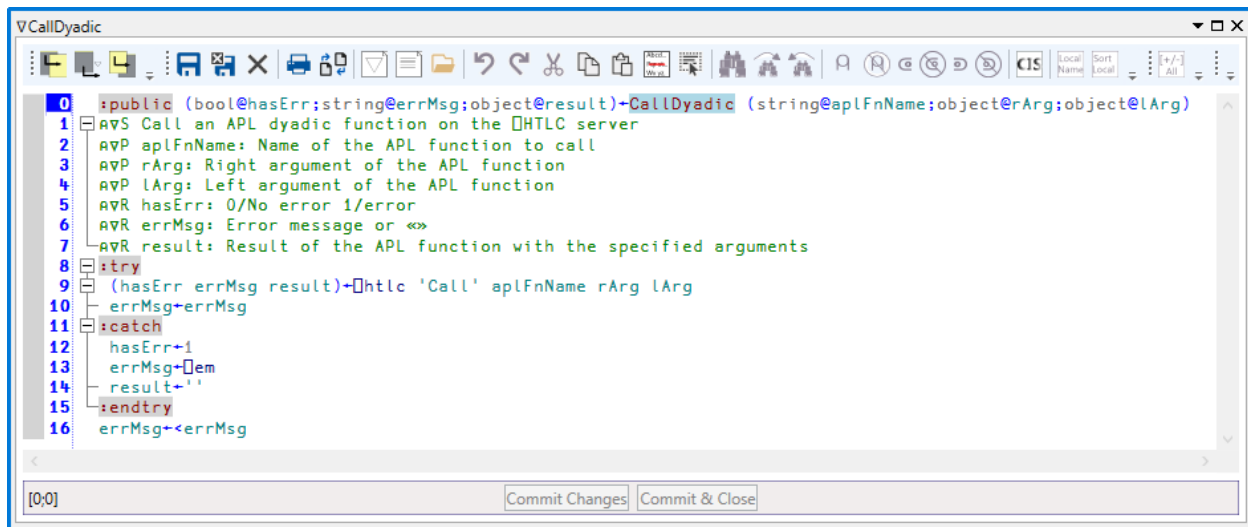
:public (bool@hasErr;string@errMsg;object@result)←CallDyadic
(string@aplFnName;object@rArg;object@lArg)
ⓂVS Call an APL dyadic function on the  HTLC server
ⓂVP aplFnName: Name of the APL function to call
ⓂVP rArg: Right argument of the APL function
ⓂVP lArg: Left argument of the APL function

```

```

⊞VR hasErr: 0/No error 1/error
⊞VR errMsg: Error message or «»
⊞VR result: Result of the APL function with the specified arguments
:try
  (hasErr errMsg result)←⊞htlc 'Call' aplFnName rArg lArg
  errMsg←errMsg
:catch
  hasErr←1
  errMsg←⊞em
  result←"
:endtry
errMsg←errMsg

```



CallMonadic

```

:public(bool@hasErr;string@errMsg;object@result)←CallMonadic
(string@aplFnName;object@rArg)
⊞VS Call an APL monadic function on the ⊞HTLC server
⊞VP aplFnName: Name of the APL function to call
⊞VP rArg: Right argument of the APL function
⊞VR hasErr: 0/No error 1/error
⊞VR errMsg: Error message or «»
⊞VR result: Result of the APL function with the specified argument

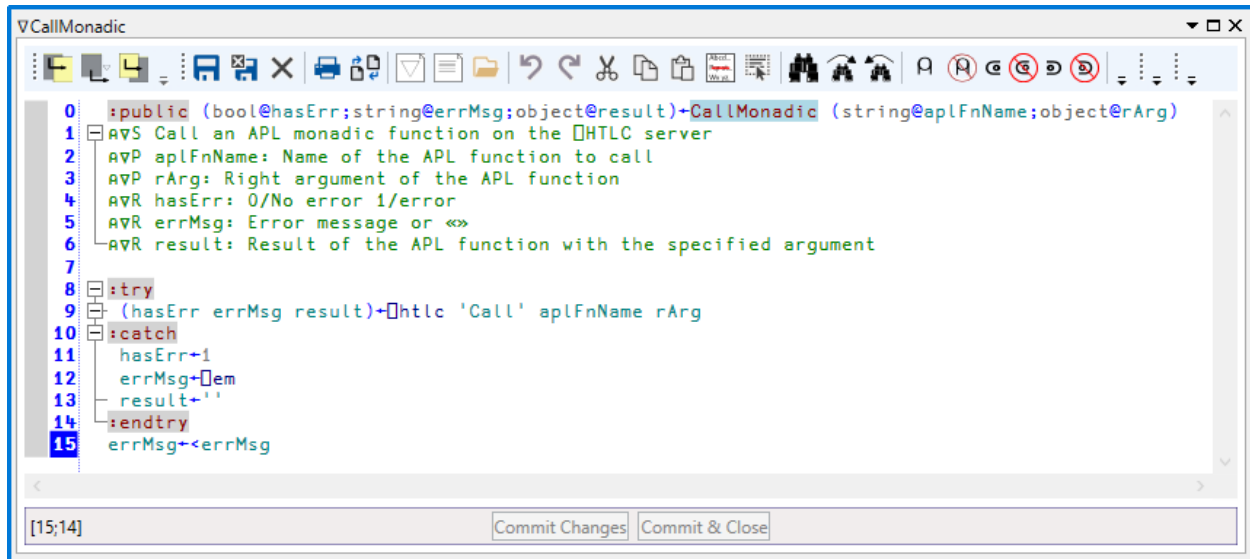
:try
  (hasErr errMsg result)←⊞htlc 'Call' aplFnName rArg
:catch
  hasErr←1
  errMsg←⊞em

```

```

result<"
:entry
errMsg<<errMsg

```



CallNiladic

```

:public (bool@hasErr;string@errMsg;object@result)←CallNiladic string@aplFnName
⍝S Call an APL niladic function on the HTLC server
⍝P aplFnName: Name of the APL function to call
⍝R hasErr: 0/No error 1/error
⍝R errMsg: Error message or «»
⍝R result: Result of the APL function
:try
(hasErr errMsg result)←⍎htlc 'Call' aplFnName
:catch
hasErr←1
errMsg←⍎em
result<"
:entry
errMsg<<errMsg

```

```

0  :public (bool@hasErr;string@errMsg;object@result)+CallNiladic string@aplFnName
1  |
2  |  ⑈VS Call an APL niladic function on the □HTLC server
3  |  ⑈VP aplFnName: Name of the APL function to call
4  |  ⑈VR hasErr: 0/No error 1/error
5  |  ⑈VR errMsg: Error message or «»
6  |  ⑈VR result: Result of the APL function
7  |  :try
8  |  |  (hasErr errMsg result)←□htlc 'Call' aplFnName
9  |  |  :catch
10 |  |  |  hasErr←1
11 |  |  |  errMsg←□em
12 |  |  |  result←''
13 |  |  :endtry
14 |  |  errMsg←<errMsg

```

[0;0] Commit Changes Commit & Close

Exec

```

:public (bool@hasErr;string@errMsg;object@result)←Exec string@aplStmt
⑈VS Execute an APL statement on the □HTLC server
⑈VP aplStmt: APL executable statement
⑈VR hasErr: 0/No error 1/error
⑈VR errMsg: Error message or «»
⑈VR result: Result of executing the specified APL statement

:try
(hasErr errMsg result)←□htlc 'Exec' aplStmt
result←result
:catch
hasErr←1
errMsg←□em
result←"
:endtry
errMsg←<errMsg

```

```

0  :public (bool@hasErr;string@errMsg;object@result)+Exec string@aplStmt
1  |
2  |  AVS Execute an APL statement on the HTLC server
3  |  AVP aplStmt: APL executable statement
4  |  AVR hasErr: 0/No error 1/error
5  |  AVR errMsg: Error message or «»
6  |  AVR result: Result of executing the specified APL statement
7  |
8  |  :try
9  |  |  (hasErr errMsg result)+Htlc 'Exec' aplStmt
10 |  |  result←result
11 |  |
12 |  |  :catch
13 |  |  |  hasErr←1
14 |  |  |  errMsg←«»
15 |  |  |  result←«»
16 |  |  :endtry
17 |  |  errMsg←errMsg

```

[15;14] Commit Changes Commit & Close

GetClientId

```

:public string@clientId←GetClientId
@VS Get the client id
@VR clientId

clientId←«»Htlc 'GetClientId'

```

```

0  :public string@clientId←GetClientId
1  |
2  |  AVS Get the client id
3  |  AVR clientId
4  |
5  |  clientId←«»Htlc 'GetClientId'

```

[0;0] Commit Changes Commit & Close

GetServerLog

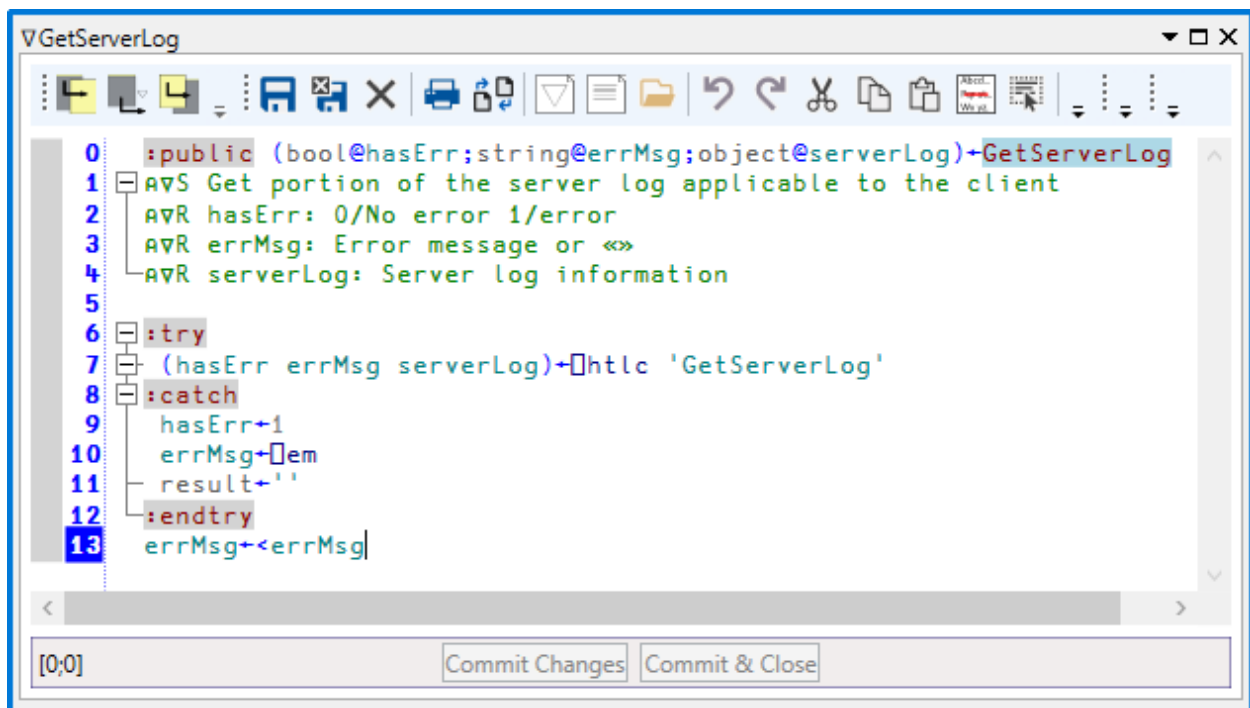
```

:public (bool@hasErr;string@errMsg;object@serverLog)←GetServerLog
@VS Get portion of the server log applicable to the client
@VR hasErr: 0/No error 1/error

```

⌚ VR errMsg: Error message or «»
 ⌚ VR serverLog: Server log information

```
:try
  (hasErr errMsg serverLog)←□htlc 'GetServerLog'
:catch
  hasErr←1
  errMsg←□em
  result←''
:endtry
errMsg←<errMsg
```



GetSysVar

```
:public (bool@hasErr;string@errMsg;object@sysVarVal)←GetSysVar string@sysVarName
⌚ VS Obtain the value of a system variable on the □HTLS server
⌚ VP sysVarName: Name of the system variable
⌚ VR hasErr: 0/No error 1/error
⌚ VR errMsg: Error message or «»
⌚ VR sysVarVal:
:try
  (hasErr errMsg sysVarVal)←□htlc 'GetSysVar' sysVarName
:catch
  hasErr←1
  errMsg←□em
```

```

sysVarVal<"
:endtry
errMsg<<errMsg

```

The screenshot shows a code editor window titled "GetSysVar". The code is as follows:

```

0  :public (bool@hasErr;string@errMsg;object@sysVarVal)-GetSysVar string@sysVarName
1  -AVS Obtain the value of a system variable on the HTLS server
2  -AVP sysVarName: Name of the system variable
3  -AVR hasErr: 0/No error 1/error
4  -AVR errMsg: Error message or «»
5  -AVR sysVarVal:
6  :try
7  (hasErr errMsg sysVarVal)-htlc 'GetSysVar' sysVarName
8  :catch
9  hasErr+1
10 errMsg+<em
11 sysVarVal+<"
12 :endtry
13 errMsg<<errMsg

```

The status bar at the bottom shows "[13;15]" and buttons for "Commit Changes" and "Commit & Close".

GetTimeout

```

:public int@timeout<GetTimeout
@VS Get client-side timeout
@VR timeout (milliseconds)

timeout<-htlc 'Timeout'

```

The screenshot shows a code editor window titled "GetTimeout". The code is as follows:

```

0  :public int@timeout-GetTimeout
1  -AVS Get client-side timeout
2  -AVR timeout (milliseconds)
3
4  timeout<-htlc 'Timeout'

```

The status bar at the bottom shows "[4;23]" and buttons for "Commit Changes" and "Commit & Close".

GetUrl

```

:public string@url<GetUrl

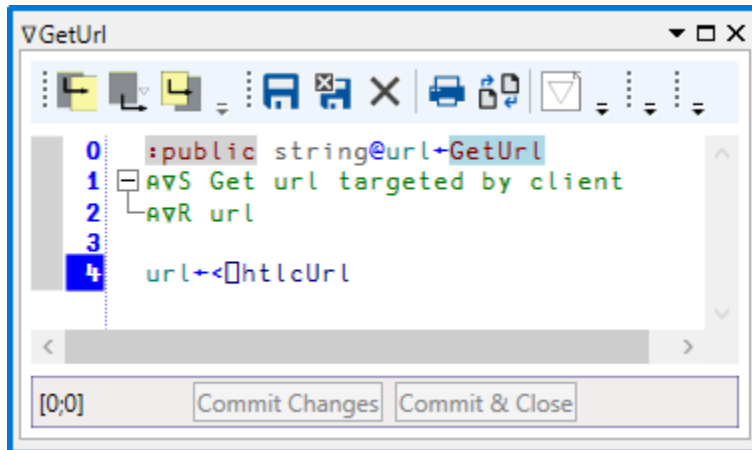
```

```

⌈⌋VS Get url targeted by client
⌈⌋VR url

url←<⌈htlcUrl

```



GetVar

```

:public (bool@hasErr;string@errMsg;object@varValue)←GetVar string@varName
⌈⌋VS Get value of a variable on the ⌈⌋HTLS server
⌈⌋VP varName: Name of the variable
⌈⌋VR hasErr: 0/No error 1/error
⌈⌋VR errMsg: error message or «»
⌈⌋VR result: Value of the APL variable
:try
(hasErr errMsg varValue)←⌈⌋htlc 'GetVar' varName
:catch
hasErr←1
errMsg←⌈em
result←"
:endtry
errMsg←<errMsg

```

The screenshot shows the APL IDE window titled 'VGetVar'. The code editor contains the following code:

```

0  :public (bool@hasErr;string@errMsg;object@varValue)←GetVar string@varName
1  ⍉VS Get value of a variable on the ⍉HTLS server
2  ⍉VP varName: Name of the variable
3  ⍉VR hasErr: 0/No error 1/error
4  ⍉VR errMsg: error message or «»
5  ⍉VR result: Value of the APL variable
6  :try
7  (hasErr errMsg varValue)←⍉htlc 'GetVar' varName
8  :catch
9  hasErr←1
10 errMsg←⍉em
11 result←''
12 :endtry
13 errMsg←errMsg

```

At the bottom of the window, there is a status bar showing '[13;14]' and two buttons: 'Commit Changes' and 'Commit & Close'.

GetVisible

```

:public (bool@hasErr;string@errMsg;bool@visible)←GetVisible
⍉VS Get the visible state of the ⍉HTLS server
⍉VR hasErr: 0/No error 1/error
⍉VR errMsg: Error message or «»
⍉VR visible: 0/Not 1/Is visible
:try
(hasErr errMsg result)←⍉htlc 'Visible'
:catch
hasErr←1
errMsg←⍉em
result←''
:endtry
errMsg←errMsg

```

```

0 :public (bool@hasErr;string@errMsg;bool@visible)->GetVisible
1 AVS Get the visible state of the HTLS server
2 AVR hasErr: 0/No error 1/error
3 AVR errMsg: Error message or «»
4 AVR visible: 0/Not 1/Is visible
5 :try
6 (hasErr errMsg result)-<[htlc 'Visible'
7 :catch
8 hasErr+1
9 errMsg+em
10 result+''
11 :endtry
12 errMsg+errMsg|

```

[12;14] Commit Changes Commit & Close

Help

```

:public string[]@result<-Help
@VS Get the HTLC documentation summary
@VR result
result<-[htlc '?'

```

```

0 :public string[]@result<-Help
1 AVS Get the HTLC documentation summary
2 AVR result
3 result<-[htlc '?'

```

[3;18] Commit Changes Commit & Close

LoadWs

```

:public (bool@hasErr;string@errMsg;object@result)<-LoadWs string@wsToLoad
@VS
@VP wsToLoad: Full path to the APL64 workspace to load
@VR hasErr: 0/No error 1/error
@VR errMsg: Error message or «»
@VR result:

```

```

:try
  (hasErr errMsg result)←⎕htlc 'LoadWs' wsToLoad
:catch
  hasErr←1
  errMsg←⎕em
  result←''
:endtry
errMsg←<errMsg

```

```

VLoadWs
0  :public (bool@hasErr;string@errMsg;object@result)←LoadWs string@wsToLoad
1  ⎕VS
2  ⎕VP wsToLoad: Full path to the APL64 workspace to load
3  ⎕VR hasErr: 0/No error 1/error
4  ⎕VR errMsg: Error message or «»
5  ⎕VR result:
6  :try
7  (hasErr errMsg result)←⎕htlc 'LoadWs' wsToLoad
8  :catch
9  hasErr←1
10 errMsg←⎕em
11 result←''
12 :endtry
13 errMsg←<errMsg

```

[13;14] Commit Changes Commit & Close

SetSysVar

```

:public (bool@hasErr;string@errMsg)←SetSysVar
(string@sysVarName;object@sysVarValue)
⌐VS Set the value of a system variable on the ⎕HTLS server
⌐VP sysVarName: Name of the system variable
⌐VP sysVarValue: New value of the system variable
⌐VR hasErr: 0/No error 1/error
⌐VR errMsg: Error message or «»
:try
  (hasErr errMsg)←⎕htlc 'SetSysVar' sysVarName sysVarValue
:catch
  hasErr←1
  errMsg←⎕em
:endtry
errMsg←<errMsg

```

```

0  :public (bool@hasErr;string@errMsg)+SetSysVar (string@sysVarName;object@sysVarValue)
1  |
2  |  AVS Set the value of a system variable on the HTLS server
3  |  AVP sysVarName: Name of the system variable
4  |  AVP sysVarValue: New value of the system variable
5  |  AVR hasErr: 0/No error 1/error
6  |  AVR errMsg: Error message or «»
7  |  :try
8  |  |  (hasErr errMsg)+htlc 'SetSysVar' sysVarName sysVarValue
9  |  |  :catch
10 |  |  hasErr+1
11 |  |  errMsg+em
12 |  |  :endtry
13 |  errMsg+<errMsg

```

[0:0] Commit Changes Commit & Close

SetTimeout

```

:public int@priorTimeout<-SetTimeout int@newTimeout
@VS Set client-side timeout
@VP newTimeout: milliseconds
@VR priorTimeout: milliseconds
priorTimeout<- htlc 'Timeout' newTimeout

```

```

0  :public int@priorTimeout<-SetTimeout int@newTimeout
1  |
2  |  AVS Set client-side timeout
3  |  AVP newTimeout: milliseconds
4  |  AVR priorTimeout: milliseconds
5  |  priorTimeout<- htlc 'Timeout' newTimeout

```

[4:40] Commit Changes Commit & Close

SetUrl

```

:public SetUrl string@url
@VS Set the url of the HTLS server targeted by the client
@VP url
htlcUrl<-url

```

The screenshot shows a code editor window titled 'VSetUrl'. The code is as follows:

```

0  :public SetUrl string@url
1  AVS Set the url of the HTLS server targeted by the client
2  AP url
3  HTLCUrl←url

```

The status bar at the bottom shows the cursor position [0;0] and two buttons: 'Commit Changes' and 'Commit & Close'.

SetVar

⌈VS Set the value of an APL variable on the HTLS server
 ⌈VP varName: Name of the variable
 ⌈VP varValue: New value of the variable
 ⌈VR hasErr: 0/No error 1/error
 ⌈VR errMsg: Error message or «»
 :try
 (hasErr errMsg)←HTLC 'SetVar' varName varValue
 :catch
 hasErr←1
 errMsg←em
 :endtry
 errMsg←<errMsg

The screenshot shows a code editor window titled 'VSetVar'. The code is as follows:

```

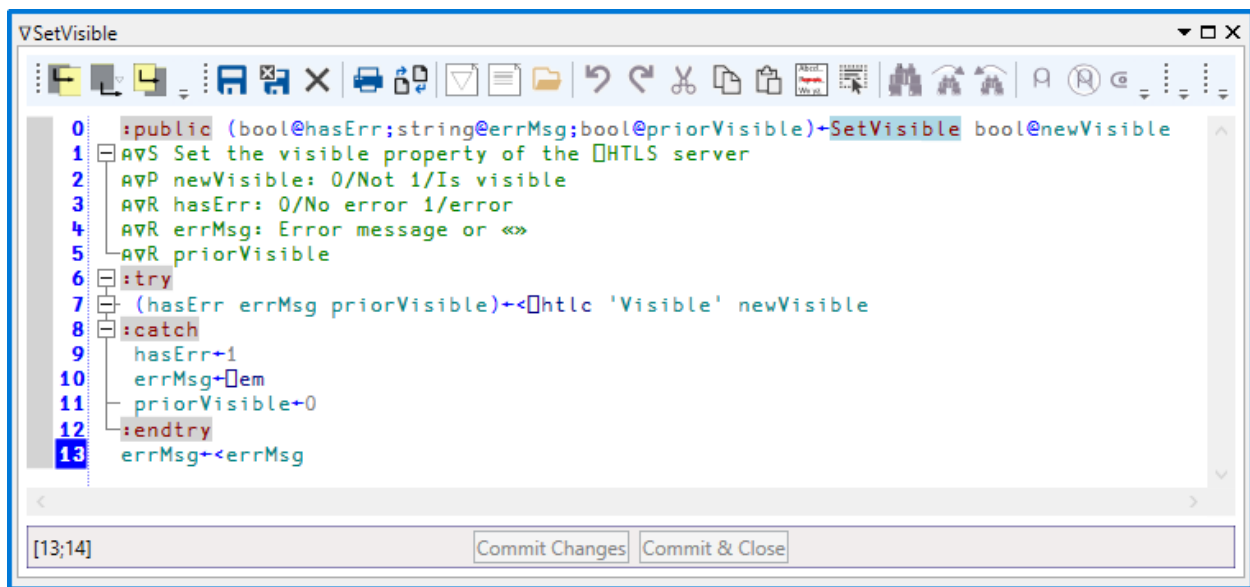
0  :public (bool@hasErr;string@errMsg)+SetVar (string@varName;object@varValue)
1  AVS Set the value of an APL variable on the HTLS server
2  AP varName: Name of the variable
3  AP varValue: New value of the variable
4  AVR hasErr: 0/No error 1/error
5  AVR errMsg: Error message or «»
6  :try
7  (hasErr errMsg)←HTLC 'SetVar' varName varValue
8  :catch
9  hasErr←1
10 errMsg←em
11 :endtry
12 errMsg←<errMsg

```

The status bar at the bottom shows the cursor position [12;14] and two buttons: 'Commit Changes' and 'Commit & Close'.

SetVisible

```
:public (bool@hasErr;string@errMsg;bool@priorVisible)<-SetVisible bool@newVisible
⌈VS Set the visible property of the □ HTLS server
⌈VP newVisible: 0/Not 1/Is visible
⌈VR hasErr: 0/No error 1/error
⌈VR errMsg: Error message or «»
⌈VR priorVisible
:try
  (hasErr errMsg priorVisible)<<□htlc 'Visible' newVisible
:catch
  hasErr<1
  errMsg<□em
  priorVisible<0
:endtry
errMsg<<errMsg
```



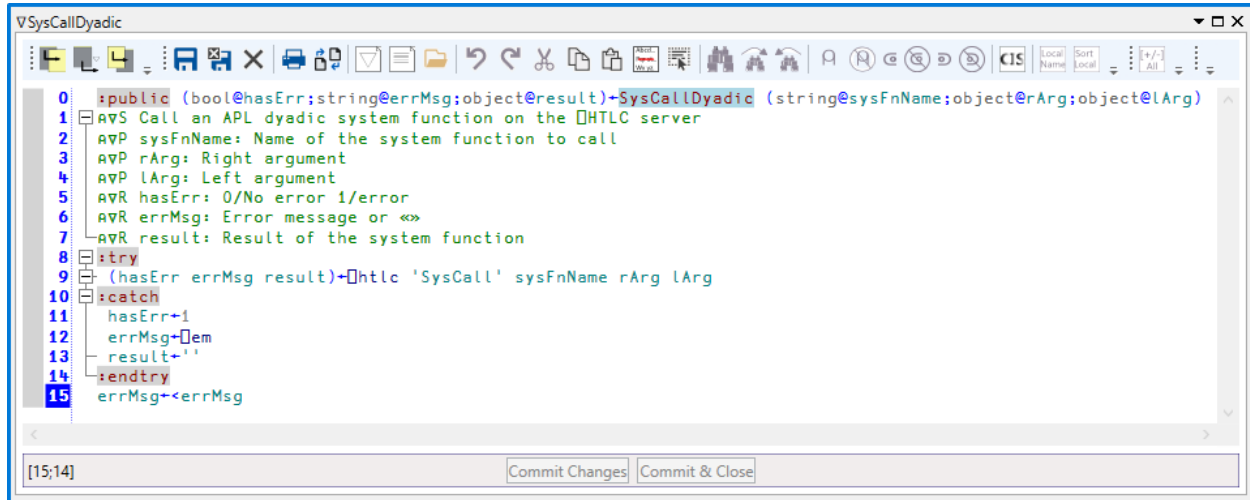
SysCallDyadic

```
:public (bool@hasErr;string@errMsg;object@result)<-SysCallDyadic
(string@sysFnName;object@rArg;object@lArg)
⌈VS Call an APL dyadic system function on the □ HTLC server
⌈VP sysFnName: Name of the system function to call
⌈VP rArg: Right argument
⌈VP lArg: Left argument
⌈VR hasErr: 0/No error 1/error
⌈VR errMsg: Error message or «»
⌈VR result: Result of the system function
:try
```

```

(hasErr errMsg result)←⊞htlc 'SysCall' sysFnName rArg lArg
:catch
hasErr←1
errMsg←⊞em
result←"
:endtry
errMsg←<errMsg

```

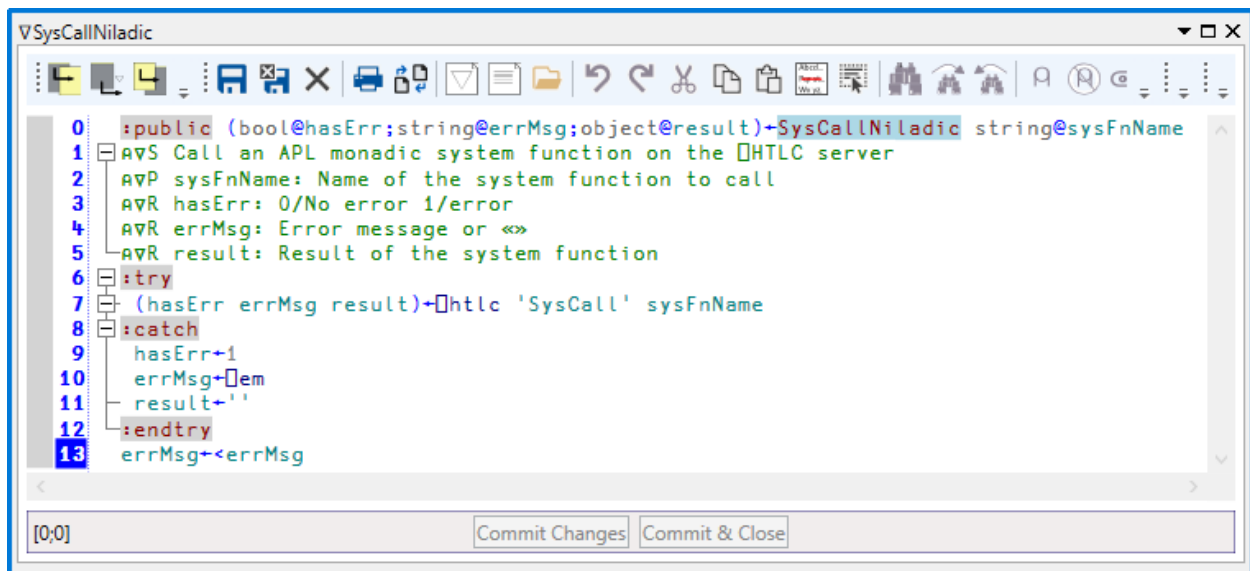


SysCallMonadic

```

:public(bool@hasErr;string@errMsg;object@result)←SysCallMonadic
(string@sysFnName;object@rArg)
⊞VS Call an APL niladic system function on the ⊞HTLC server
⊞VP sysFnName: Name of the system function
⊞VP rArg: Right argument of the system function
⊞VR hasErr: 0/No error 1/error
⊞VR errMsg: Error message or «»
⊞VR result: Result of the system function
:try
(hasErr errMsg result)←⊞htlc 'SysCall' sysFnName rArg
:catch
hasErr←1
errMsg←⊞em
result←"
:endtry
errMsg←<errMsg

```

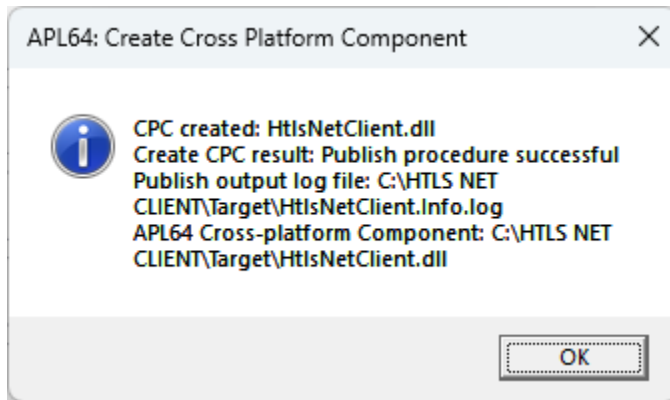
```
0 :public (bool@hasErr;string@errMsg;object@result)←SysCallNiladic string@sysFnName
1 ⍷S Call an APL monadic system function on the ⍷HTLC server
2 ⍷P sysFnName: Name of the system function to call
3 ⍷R hasErr: 0/No error 1/error
4 ⍷R errMsg: Error message or «»
5 ⍷R result: Result of the system function
6 :try
7   (hasErr errMsg result)←⍷htlc 'SysCall' sysFnName
8 :catch
9   hasErr←1
10  errMsg←⍷em
11  result←''
12 :endtry
13 errMsg←<errMsg
```

[0;0] Commit Changes Commit & Close

SysCommand

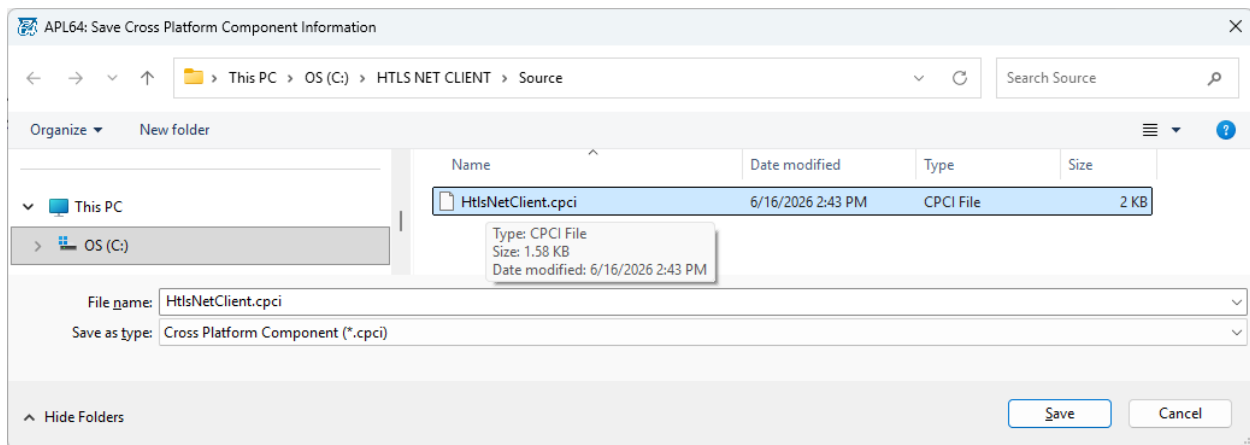
```
:public (bool@hasErr;string@errMsg)←SysCommand string@sysCommandStmt
⍷S Execute an APL system command on the ⍷HTLS server
⍷P sysCommandStmt
⍷R hasErr: 0/No error 1/error
⍷R errMsg: Error message or «»
:try
  (hasErr errMsg)←⍷htlc 'SysCommand' sysCommandStmt
:catch
  hasErr←1
  errMsg←⍷em
:endtry
errMsg←<errMsg
```


Click the CPC creation dialog 'Create' button



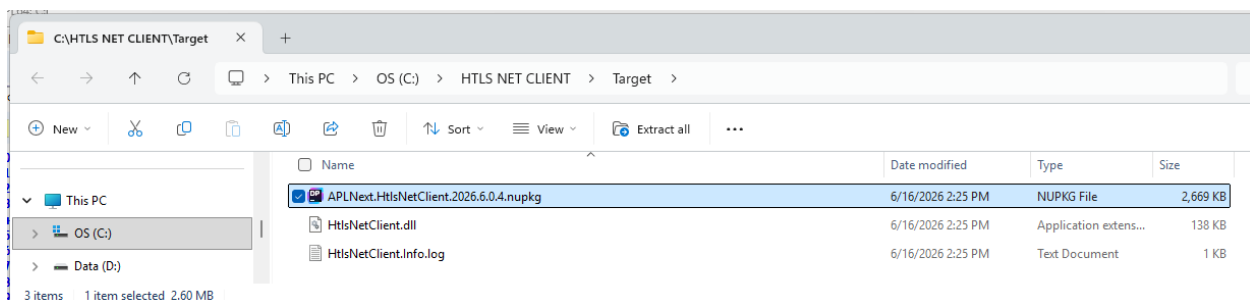
Save the CPC Creation Dialog Info

Click the CPC creation dialog 'Save CPC Info' button to preserve the CPC configuration for future use.

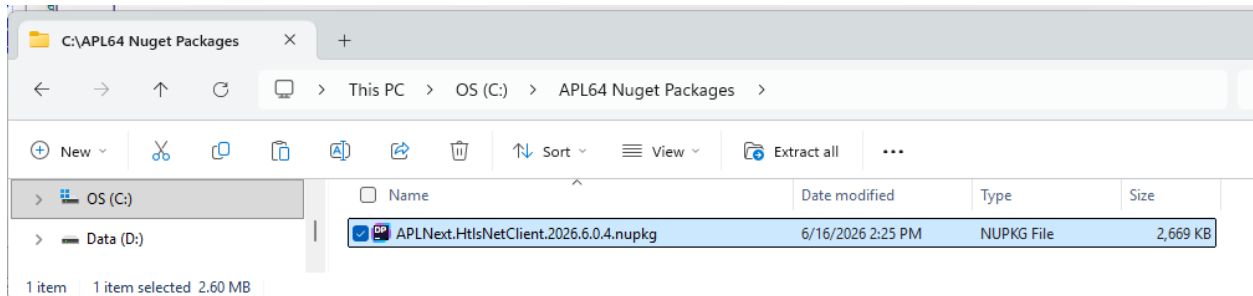


Copy the resulting Nuget package from the Target folder to the Nuget package publish folder:

From:

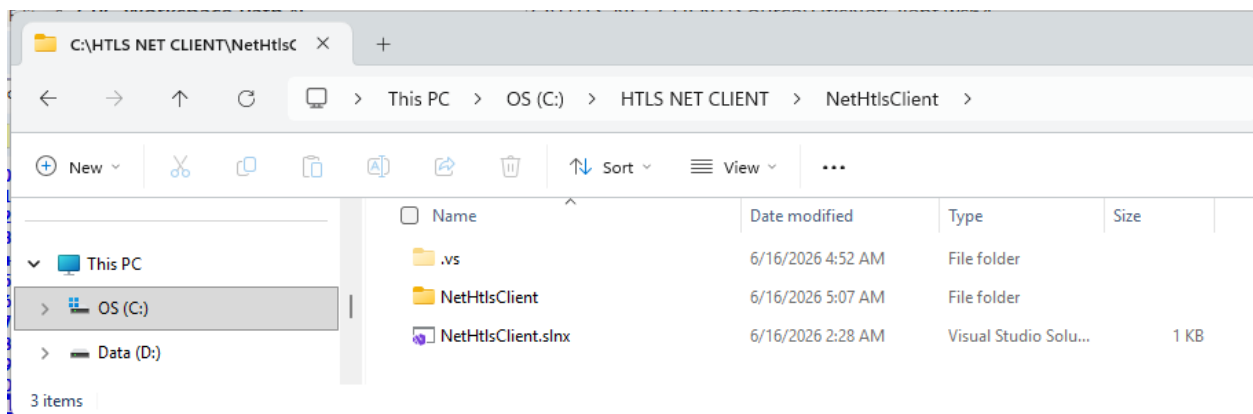


To:

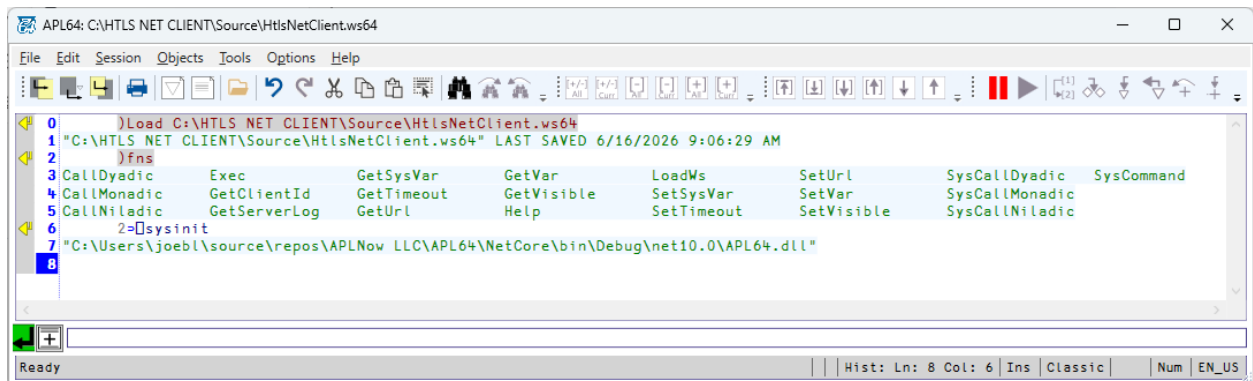


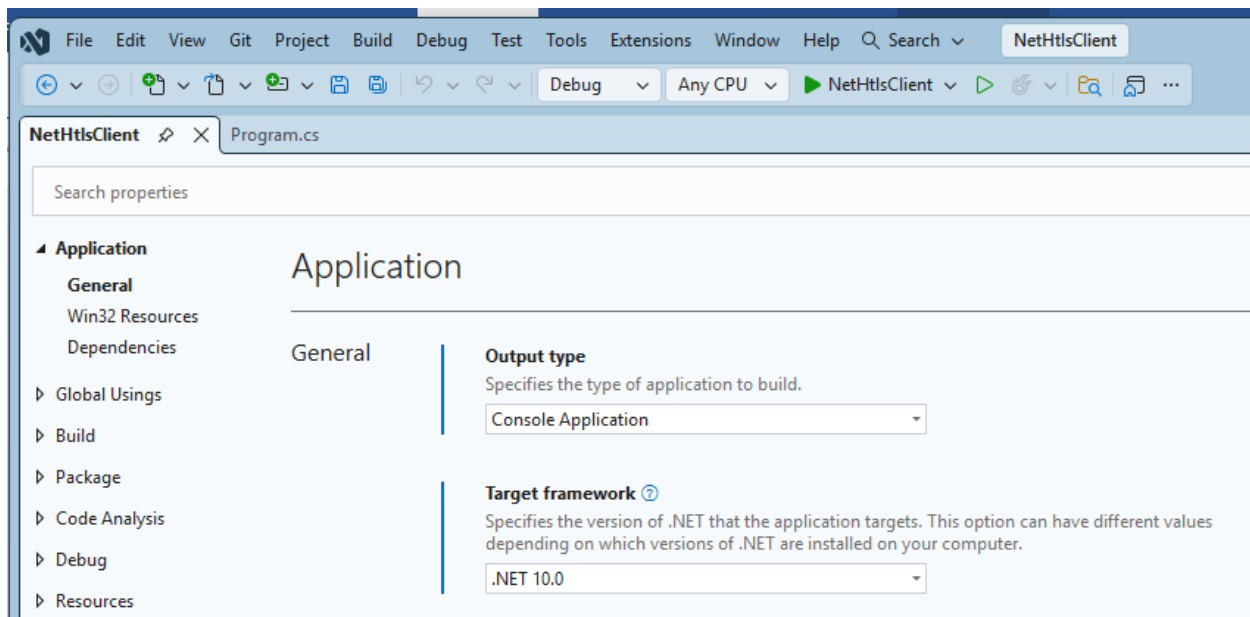
Create a .Net Project to use the CPC Nuget Package

Using Visual Studio create the NetHttpsClient C# console project in the C:\HTLS NET CLIENT\NetHttpsClient folder:



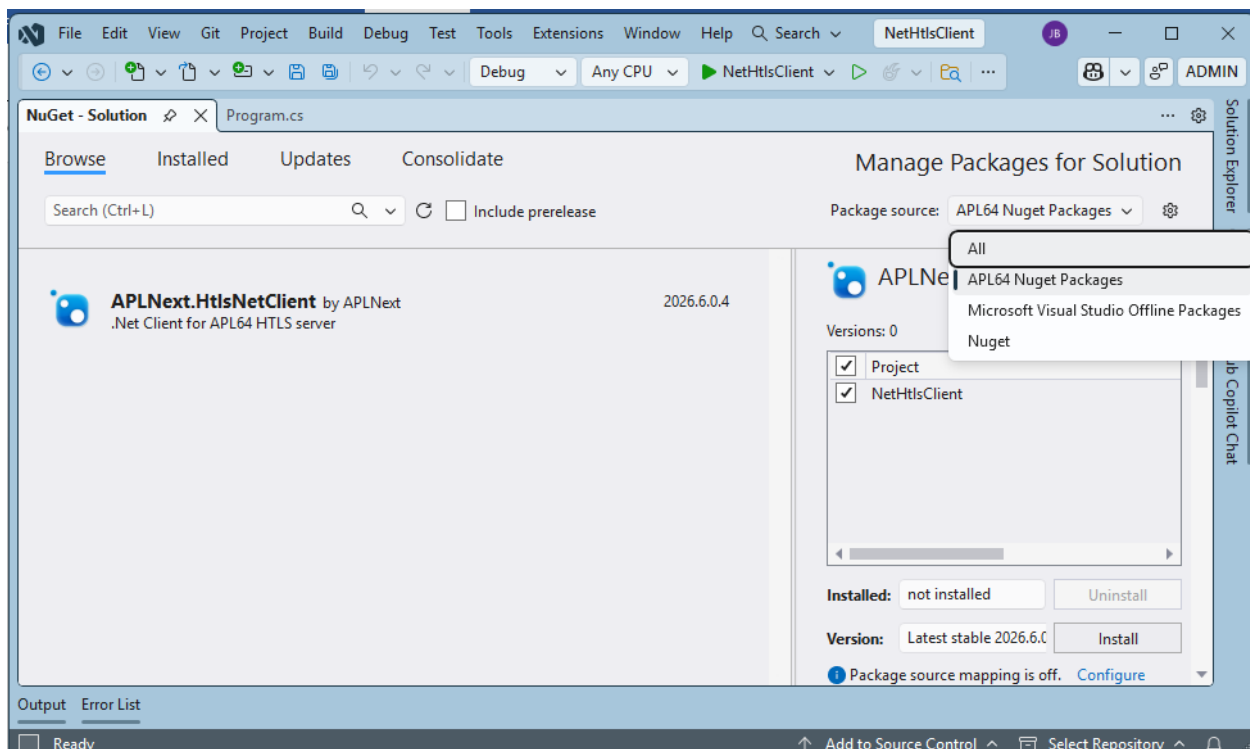
This project is targeted to the current APL64 .Net version which can be checked using:





Install the CPC Nuget Package into the .Net Project

Use Tools > Nuget Package Manager > Manage Nuget Packages for Solution dialog to install the APL64 CPC Nuget package into the solution.



Enter the C# source code

This source code illustrates the use of some ☐ HTLC actions in the APL64 CPC Nuget package, including SetUrl, SysCallMonadic, CallMonadic, Exec, SetVar, GetVar, SetSysVar and GetSysVar:

```
using HtlsNetClient;
namespace NetHtlsClient
{
    internal class Program
    {
        static void Main(string[] args)
        {
            //Note: Start the server before running this client code. The server can be started by
            //executing ☐htls 'Listen' 'http://localhost'

            var url = "http://localhost";
            var client = new ClientClass();
            client.SetUrl(url);

            (bool hasErr, string errMsg, object res) = client.SysCallMonadic("DEF", new string[] {
            "Z←Add10 X", "Z←10+X" });
            if (hasErr)
                Console.WriteLine($"Error: {errMsg}");
            else if (res.GetType() != typeof(System.Char[]))
                Console.WriteLine($"Function Add10 could not be defined on server");
            else
            {
                Console.WriteLine($"Function Add10 defined on server");
                (hasErr, errMsg, res) = client.CallMonadic("Add10", 1000);
                if (hasErr)
                    Console.WriteLine($"Error: {errMsg}");
                else
                {
                    Console.WriteLine($"Call function Add10 1000 with result: {res}");
                    (hasErr, errMsg, res) = client.Exec("0.1 + ι3");
                    if (hasErr)
                        Console.WriteLine($"Error: {errMsg}");
                    else
                    {
                        Console.WriteLine($"Execute 0.1 + ι3:");
                        var dV = (double[])res;
                        if (dV == null)
                            Console.WriteLine($"Result is not a double array");
```

```

else
{
    for (int i = 0; i < dV.Length; i++)
    {
        Console.WriteLine(dV[i]);
    }
    (hasErr, errMsg) = client.SetVar("MyVar", new int[] { 1, 2, 3 });
    if (hasErr)
        Console.WriteLine($"Error: {errMsg}");
    else
    {
        Console.WriteLine($"Variable MyVar set on server");
        (hasErr, errMsg, res) = client.GetVar("MyVar");
        if (hasErr)
            Console.WriteLine($"Error: {errMsg}");
        else
        {
            Console.WriteLine($"Variable MyVar retrieved from server:");
            var iV = (int[])res;
            if (iV == null)
                Console.WriteLine($"Result is not an int array");
            else
            {
                for (int i = 0; i < iV.Length; i++)
                {
                    Console.WriteLine(iV[i]);
                }
                (hasErr, errMsg) = client.SetSysVar("IO", 0);
                if (hasErr)
                    Console.WriteLine($"Error: {errMsg}");
                else
                {
                    Console.WriteLine($"System variable IO set to 0 on server");
                    (hasErr, errMsg, res) = client.GetSysVar("IO");
                    if (hasErr)
                        Console.WriteLine($"Error: {errMsg}");
                    else
                        Console.WriteLine($"System variable IO retrieved from server with
value: {res}");
                }
            }
        }
    }
}

```



NetHtlClient - Program.cs*

Program.cs* X

NetHtlClient

NetHtlClient.Program

Main(string[] args)

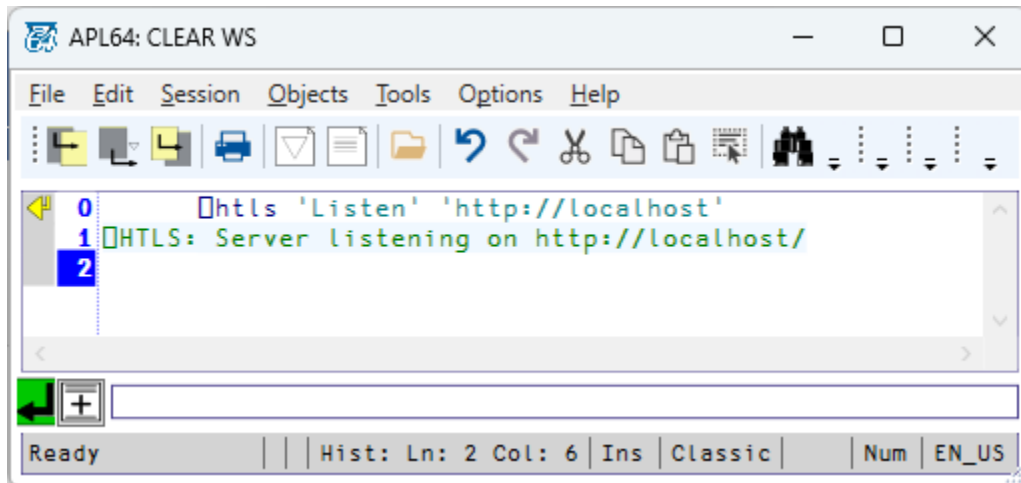
```
1 using HtlClient;
2 namespace NetHtlClient
3 {
4     class Program
5     {
6         static void Main(string[] args)
7         {
8             //Note: Start the server before running this client code. The server can be started by executing HtlClient 'listen' 'http://localhost'
9
10            var url = "http://localhost";
11            var client = new ClientClass();
12            client.SetUrl(url);
13
14            (bool hasErr, string errMsg, object res) = client.SysCallMonadic("DEF", new string[] { "2+Add10 X", "2+10*X" });
15            if (hasErr)
16                Console.WriteLine($"Error: {errMsg}");
17            else if (res.GetType() != typeof(System.Char[]))
18                Console.WriteLine($"Function Add10 could not be defined on server");
19            else
20            {
21                Console.WriteLine($"Function Add10 defined on server");
22                (hasErr, errMsg, res) = client.CallMonadic("Add10", 1000);
23                if (hasErr)
24                    Console.WriteLine($"Error: {errMsg}");
25                else
26                {
27                    Console.WriteLine($"Call function Add10 1000 with result: {res}");
28                    (hasErr, errMsg, res) = client.Exec("0.1 + i3");
29                    if (hasErr)
30                        Console.WriteLine($"Error: {errMsg}");
31                    else
32                    {
33                        Console.WriteLine($"Execute 0.1 + i3:");
34                        var dV = (double[])res;
35                        if (dV == null)
36                            Console.WriteLine($"Result is not a double array");
37                        else
38                        {
39                            for (int i = 0; i < dV.Length; i++)
40                            {
41                                Console.WriteLine(dV[i]);
42                            }
43                            (hasErr, errMsg) = client.SetVar("MyVar", new int[] { 1, 2, 3 });
44                            if (hasErr)
45                                Console.WriteLine($"Error: {errMsg}");
46                            else
47                            {
48                                Console.WriteLine($"Variable MyVar set on server");
49                                (hasErr, errMsg, res) = client.GetVar("MyVar");
50                                if (hasErr)
51                                    Console.WriteLine($"Error: {errMsg}");
52                                else
53                                {
54                                    Console.WriteLine($"Variable MyVar retrieved from server:");
55                                    var iV = (int[])res;
56                                    if (iV == null)
57                                        Console.WriteLine($"Result is not an int array");
58                                    else
59                                    {
60                                        for (int i = 0; i < iV.Length; i++)
61                                        {
62                                            Console.WriteLine(iV[i]);
63                                        }
64                                        (hasErr, errMsg) = client.SetSysVar("i0", 0);
65                                        if (hasErr)
66                                            Console.WriteLine($"Error: {errMsg}");
67                                        else
68                                        {
69                                            Console.WriteLine($"System variable i0 set to 0 on server");
70                                            (hasErr, errMsg, res) = client.GetSysVar("i0");
71                                            if (hasErr)
72                                                Console.WriteLine($"Error: {errMsg}");
73                                            else
74                                                Console.WriteLine($"System variable i0 retrieved from server with value: {res}");
75                                        }
76                                    }
77                                }
78                            }
79                        }
80                    }
81                }
82            }
83        }
84    }
85 }
```

56 % No issues found Ln: 85, Ch: 2 SPC CRLF UTF-8 with BOM

Start an APL64 instance and use ☐ HTLS to create an HTLS server

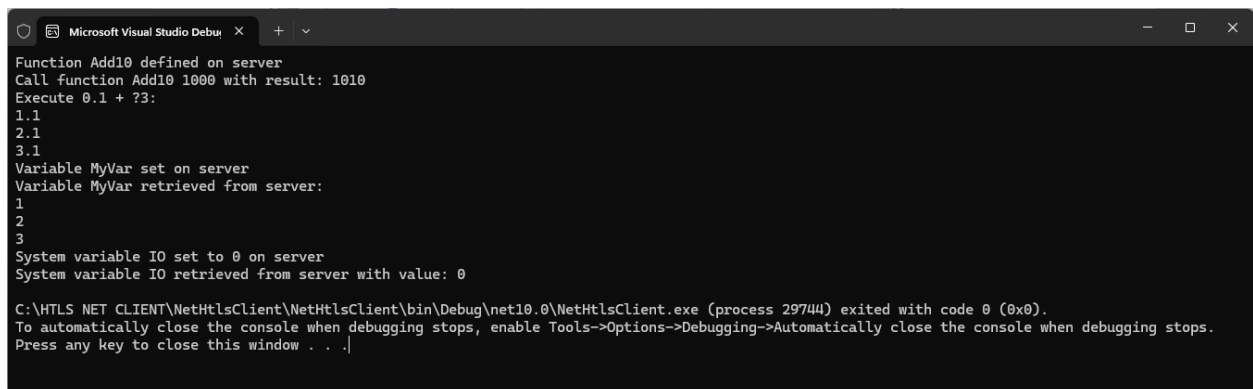
An ☐ HTLS server can be started manually from an instance of APL64 Developer version, APL64 WRE (Windows Runtime Executable) or APL64 CPC (Cross-platform Component).

```
☐htls 'Listen' 'http://localhost'
```



Run the console application

Use F5 in Visual Studio to run the .Net C# console project using the APL64 HtlsNetClient CPC Nuget package:



Comparing ☐ HTLC/HTLS and ☐ WSE

Analogous Client and Server Environments

Both ☐ HTLC/HTLS and ☐ WSE create client and server environments.

□HTLC/HTLS uses two independent instances of APL64 for the separate client and server environments. The □HTLS server is an ‘out-of-process’ server, since it is not running within the memory space of the □HTLC client instance.

When □HTLS is used in the APL64 development environment, the □HTLS server can be made visible and can be debugged. The □HTLS server cannot be made visible or debugged in an APL64 WRE or CPC. Appropriate end-user security should be considered when the □HTLS server is used in an APL64 WRE or CPC.

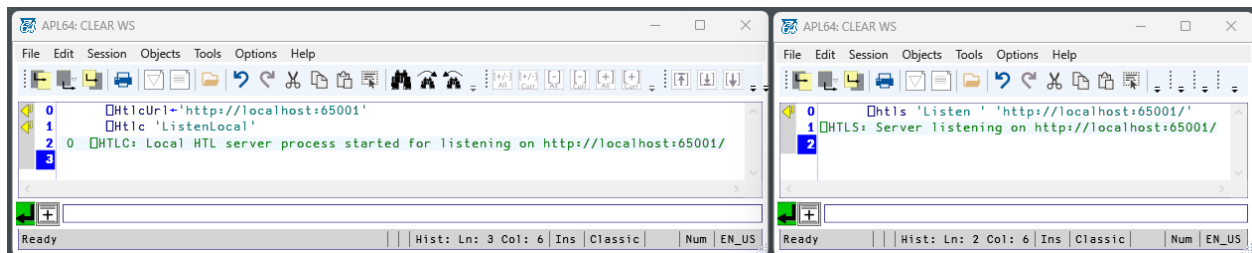
□WSE (Workspace Engine) uses a single instance of APL64 for the separate client and server environments. The □WSE is an ‘in-process’ server, since it is running within the memory space of the APL64 instance which executed the □WSE ‘Create’ action.

The □WSE server cannot be made visible and cannot be debugged. The APL64 □WSE system function has parallel processing actions in addition to CallAsync, which are not applicable to □HTLC/HTLS.

Creating the Server

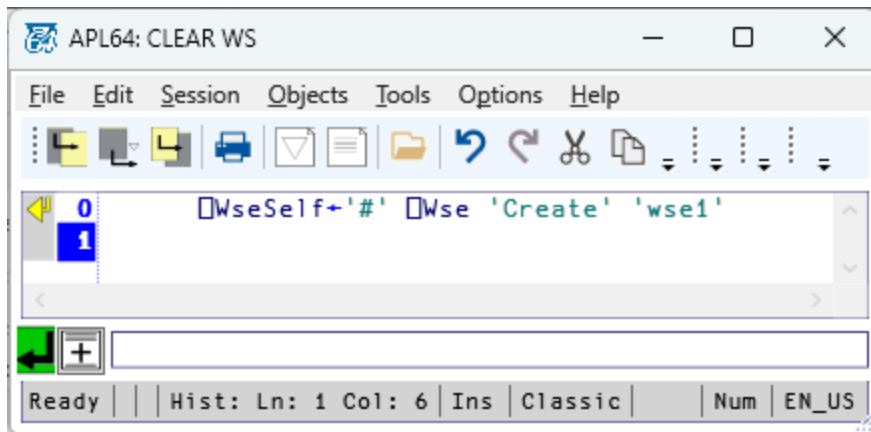
□HTLS servers are identified by distinct urls. An □HTLS server is visible by default but can be hidden using the □HTLC Visible action. An □HTLS server cannot hide itself.

```
□HtlcUrl←'http://localhost:65001'  
□Htlc 'ListenLocal'
```



□WSE servers are identified by name. An □WSE server is not visible.

```
□WseSelf←'#' □wse 'Create' 'wse1'
```



□ HTLC and □ WSE have Analogous Actions

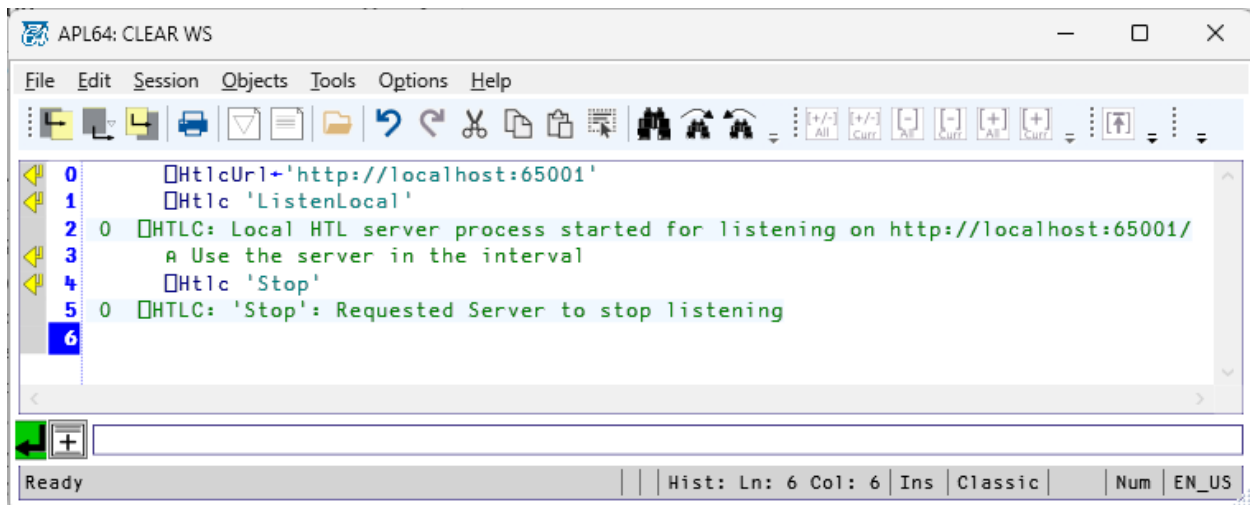
Because □ HTLC and □ WSE have analogous actions an APL64 programmer can use an □ HTLS server for visible debugging in a development environment and use a □ WSE server in the runtime environment.

Action	□ HTLC and □ WSE Argument(s)
Call	fnName [rArg] [lArg]
CallAsync	fnName [rArg] [lArg]
Exec	aplExpression
GetSysVar	sysVarName
GetVar	varName
LoadWs	wsPath
SetSysVar	sysVarName value
SetVar	varName value
SysCall	sysFnName [rArg] [lArg]
SysCommand	sysCmdName text
Variable	varName [value]

Closing the Server

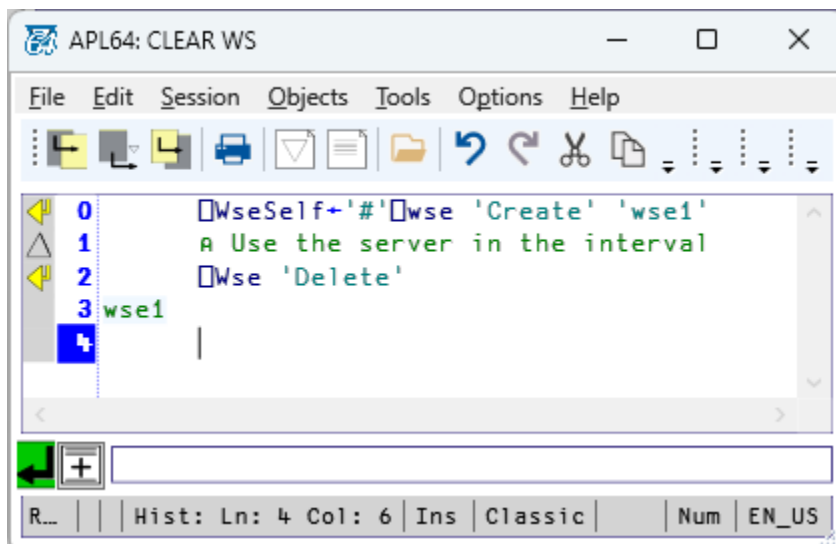
□ HTLC

☐ HtlcUrl←'http://localhost:65001'
☐ Htlc 'ListenLocal'
☒ Use the server in the interval
☐ Htlc 'Stop'



☐ WSE

☐ WseSelf←'#' ☐ wse 'Create' 'wse1'
 Ⓞ Use the server in the interval
☐ Wse 'Delete'



Multiple Instances

Multiple instances of ☐ HTLS servers are supported. The same or different instances of APL64 can listen to different urls.

Multiple instances of ☐ WSE servers are supported. Each ☐ WSE server must have a distinct name.

Performance

An ☐HTLS and a ☐WSE server use the same APL64 interpreter technology. Their performance differs because of how APL64 data is transferred between client and server.

Because an ☐HTLS server is an out of process server, communication with an ☐HTLS server uses the http data transfer protocol. Since the http protocol is designed for byte-format data, serialization of APL64 data occurs when ☐HTLC actions access an ☐HTLS server and when an ☐HTLS server responds to an ☐HTLC request. The out of process design of an ☐HTLS server is what permits debugging and visibility of the server.

A ☐WSE server is an in-process server, so there is no data transfer protocol or serialization of APL data for communication between a ☐WSE client and ☐WSE server. The in-process design of a ☐WSE server means that it cannot be debugged or made visible.

Debugging

An ☐HTLC client and a ☐WSE client can be debugged if they are used in the APL64 Developer version.

☐HTLC clients and ☐HTLS servers can be independently debugged when they are created with an APL64 Developer version instance. Only the client side of a ☐WSE client and server arrangement can be debugged.

Configuring an ☐HTLS server to listen on an HTTPS port

An https port is generally required for a server to be accepted by responsible clients. A server exposed to clients which does not use an https port will generate client-side warnings which may limit server use by a client.

Syntax: msg ← ☐HTLS 'Listen' serverHttpsUrl certificatePath certificatePassword

Besides providing the https url, certificate path and certificate password, additional setup of the server machine environment is necessary. This section of the document describes the required setup elements.

Port Forwarding (For access by external clients)

If the server is behind a router/firewall, and needs to accept external input, the Internet service router needs configuration. The Internet router configuration must be modified to permit port forwarding (synonyms: virtual server or NAT) to the HTLS server.

- Configure port forwarding rules on the router

- Forward external port input to the internal server port

Default HTTPS port is 443 which requires administrator rights, but custom ports like 8443 or 9443 may not require administrator rights.

Firewall Configuration

Windows Firewall must allow inbound traffic on the HTTPS port. The choice of access type must also be made: Private, Public access types.

Administrator Privileges

They are required when:

- Listening on privileged ports (< 1024, including port 443)
- Binding to wildcard addresses (*) or (+)
- Accessing HTTP.SYS namespace reservations

Certificate Trust and Validation

Use certificates from trusted certificate authorities such as:

- Let's Encrypt (Free)
- DigiCert
- GoDaddy

For the development and testing environment a self-signed certificate can be used.

Url Binding Permissions (http.sys on Windows)

When binding to wildcard (*) or (+) addresses without administrator privileges,

URL ACL (Access Control List) reservations may be necessary.

Wildcards can be supported:

- *(asterisk) - Strong wildcard, binds to all IP addresses
- + (plus) - Weak wildcard, allows sharing with other applications

Dns and Hostname Resolution

For external access configure DNS A record pointing to server's public IP address, e.g. api.yourdomain.com points to 203.0.113.50

For internal network access configure an internal DNS server, or add an entry to the client hosts file in the c:\Windows\System32\drivers\etc\hosts location, e.g.

192.168.1.100 points to the local DNS: api.localserver.com

Router Settings (For external Https access)

- Reserve specific internal IP for the server in the router DHCP settings
- Ensure port forwarding always points to correct device
- Assure the ISP does not block the selected port (e.g. 443)

Network Topology

Server Location	Configuration Requirements
Same workstation (localhost)	Firewall
Local network (192.168.x.y)	Firewall, optional local DNS
External access (registered dsn)	Firewall, port forwarding, registered DNS

Testing and Verification

- Certificate file exists and is readable
- Firewall allows inbound traffic on HTTPS port
- Running as administrator (if port < 1024)
- Port forwarding configured (if external access needed)
- DNS/hosts pointing to server IP address
- Certificate trusted by clients

☐ HTLS validation and exception handling

Exception	Indicates
UnauthorizedAccessException	Administrator rights likely required
FileNotFoundException	Certificate file missing or path incorrect
CryptographicException	Certificate password or invalid format
IOException	Port already in use or access denied

Validation
URL format validation
Port number range (1-65535)
Scheme validation (http or https)
Certificate file existence
Host format (localhost, IP address, wildcards)

Security Best Practices

Certificates

- Use strong passwords for .pfx files
- Restrict file permissions to minimum required
- Rotate certificates before expiration
- Use 2048-bit or higher RSA keys
- Consider ECC certificates for better performance

Network

- Use firewall rules to restrict access by IP when possible
- Disable UPnP in production environments
- Use non-standard ports to reduce automated attacks
- Implement rate limiting and DDoS protection
- Monitor logs for suspicious activity

Authentication

- Implement client certificate validation if needed
- Use strong authentication mechanisms
- Enable logging using HTLS LogFileOptions action