DYALOG

The tool of thought for software solutions

The JSON_APL Shared Object

Version 18.0

Dyalog Limited

Minchens Court, Minchens Lane
Bramley, Hampshire
RG26 5BH
United Kingdom

tel: +44 1256 830030
fax: +44 1256 830031
email: support@dyalog.com
https://www.dyalog.com

Version 18.0

Revision: 20200326_180

Contents

1 About This Document

This document describes the JSON_APL Shared Object (included with standard installations of Dyalog version 17.1 onwards). The JSON_APL Shared Object allows the user to execute APL statements in the Dyalog interpreter from any programming language that can access native shared libraries.

JSON_APL is a specific example of more general tools for wrapping APL applications as shared libraries, as documented in *APL as a Shared library*. Only the binary shared object library file and Python example are included with a standard Dyalog installation; the source code is provided as one of several samples that can be downloaded from https://github.com/dyalog/NativeLib.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog, knowledge of the APL application to be called, the mechanisms used by the client language to make calls to shared libraries (typically called the "foreign function interface") and how to create and use JSON strings in that environment.

2 Introduction

JSON_APL is an interface that allows any programming language to execute APL expressions and call APL functions. Arguments are passed and results are received as character strings in a format known as JSON (JavaScript Object Notation). For example, to sum the first 5 even numbers using APL, the caller would need to create the strings `"+/"` and `"[2,4,6,8,10]"` and invoke the function `CallJSON`, passing those strings as parameters. The result will be the string `"30"`, which is the JSON representation of the numeric scalar result.

JSON_APL is included with standard installations of Dyalog version 17.1 or later in the form of a shared library with an extension appropriate for each platform (**.dll** under Microsoft Windows, **.so** under Linux or **.dylib** under macOS). Examples of how to use it to call APL from C and Python are included in this document, and the source code can be downloaded from https://github.com/dyalog/NativeLib. The Python example (also included with standard installations of Dyalog) is only compatible with Python 3.

JSON_APL is initialised by specifying APL start-up parameters and the location of the APL code that is to be made available. It can be used to make any APL code available without changes to the APL application as long as all arguments and results can be represented in JSON. Unfortunately, JSON cannot represent multi-dimensional arrays, so these typically need to be split into vectors of vectors and re-assembled into arrays. Apart from this limitation, the vast majority of APL arrays can be represented.

3 Entry points

This chapter describes the functions used for setting the active workspace parameters, loading APL code into the workspace and accessing the APL functions using JSON. Apart from the `Initialise` function, all entry points call APL functions that have been compiled into the shared object. The source for these functions is not included in a standard installation but can be downloaded separately from https://github.com/dyalog/NativeLib. The source file **JSON_APL/JSON_APL/JSON_APL/JSON_APL.dyalog** contains the APL functions described in *Section 3.2*, which are used to execute the function calls in the workspace.

3.1 C functions

```
int LIBCALL Initialise(unsigned int runtime, unsigned int
len, wchar_t **args)
```

The `Initialise` function takes a Boolean flag `runtime` to indicate whether to use the runtime interpreter (`1`) or development interpreter (`0`). If the development interpreter is used then (on Microsoft Windows) any untrapped errors in the APL code will cause the Dyalog Development Environment to be displayed. In other scenarios, the RIDE_INIT environment variable, or 3502⌶ (manage RIDE connections) can be used to configure the interpreter to allow RIDE connections for debugging.

An array of pointers to wide character strings `args` is used to set interpreter environment variables (for example, `"MAXWS=512Mb"` or `"-Dcw"`). The integer `len` specifies how many entries are in the `args` array.

The function starts the APL interpreter with the specified configuration and returns `0` if successful.

```
int LIBCALL CallJSON(wchar_t *fname, wchar_t *in, wchar_t
*out, unsigned int len)
```

The `CallJSON` function executes APL statements and calls APL functions, passing arguments and receiving results as wide character strings in JSON format.

The wide character string `fname` is the name of the function in the APL workspace. This can be either one of the built-in functions described in *Section 3.2* or one that has been loaded into the workspace using the `LoadAPL` function.

The wide character string `in` is either:

- A JSON object, represented as a wide character string, containing:
  - A `"Function"` string of a monadic or dyadic APL function.
  - An optional `"Right"` string, numeric or list for the right argument to the function.

  - An optional `"Left"` string, numeric or list for the left argument to the dyadic function.

  For example: Sum the first five integers
  `{"Function": "+/", "Right": [1,2,3,4,5]}`
  For example: Apply a Boolean mask to a character array
  `{"Function": "/", "Left": [1,0,1,1,0], "Right": "APPLE"}`
- A wide character string representing a complete APL statement
  For example: Sum the first five integers
  `"+/⍳5"` (or `"+/\\u23739"` with explicit Unicode code points)

The wide character buffer `out` is used to store the result of the APL statement. The integer `len` specifies the length of `out` and `out` must be large enough to store the string representation of the APL statement result as otherwise the result could be truncated.

The function returns an integer error code as described in *APL as a Shared Library*.

```
int LIBCALL ExecAPL(wchar_t *statement, wchar_t *result)
```

The `ExecAPL` function calls the APL function **`Exec`** described *Section 3.2*. The wide character string `statement` is the complete APL statement to be executed. The wide character buffer `result` must be large enough to contain the text-formatted result of the APL statement as otherwise the result could be truncated.

The function returns an integer error code as described in *APL as a Shared Library*.

```
int LIBCALL GetEnv(const wchar_t *name, wchar_t *value,
size_t len)
```

The `GetEnv` function calls the APL function **`GetEnv`** described in *Section 3.2*. The wide character string `name` is the name of an interpreter environment variable (for example, `"MAXWS"`). The wide character string `value` is the returned value of the named environment variable. The size_t `len` is the size of the wide character buffer used to return the value; it must be large enough to store the string representation of that value.

3.2 APL functions

The built-in APL functions, found in **JSON_APL/JSON_APL/JSON_APL.dyalog** inside the source code for the JSON_APL Shared Object (downloadable from https://github.com/dyalog/NativeLib), are compiled into the Shared Object to make calling APL using JSON more convenient. Unless otherwise specified, if there is an error in execution there will be no result but the `CallJSON` function (see *Section 3.1*) will return an error code (as described in *APL as a Shared Library*, positive error codes are Dyalog error codes) as its result. The APL functions are described below:

```
∇ r←Load APLCode
```

The **`Load`** function brings APL code into the active workspace from a binary **.dws** workspace or an APL text source file. **`APLCode`** is a simple character vector representing the file path (relative or absolute) of the APL code source. Text source files must have one of the file extensions **.dyalog**, **.aplf**, **.aplo**, **.apln**, **.aplc** or **.apli**. The **.dyalog** extension is an

APL text source file which may contain a class, namespace, interface or standalone APL statements, while the other extensions refer to source files created by Link (https://github.com/Dyalog/link) which correspond to specific name classes as described in Table 1. The `Load` function can be used multiple times to bring multiple APL functions or objects into the workspace.

The output, `r`, is the numeric scalar `0` if the APL source file or workspace loaded successfully, and `1` otherwise.

***Table 1**: APL text source file extensions and corresponding name classes*

Name class	Extension
3	.aplf
4	.aplo
9.1	.apln
9.4	.aplc
9.5	.apli

```
∇ out←Exec APL
```

`Exec` is used to execute APL statements.

`APL` is one of the following:

- A JSON object containing a character vector property `Function` that represents the APL function statement, and optional properties `Right` and `Left` that contain the right and left arguments respectively.
- A simple character vector describing a complete APL statement.

If the function executes successfully then the result `out` is a simple character vector of the function result. Otherwise there is no result, but the `CallJSON` function (see *Section 3.1*) will return an error code (error codes are described in *APL as a Shared Library*).

```
∇ r←GetEnv n
```

The GetEnv function is a wrapper for `2 ⎕NQ '.' 'GetEnvironment' var` and is used to verify that the interpreter has been initialised correctly with the specified environment variables.

`n` is a simple character vector giving the name of an environment variable (for example, `'MAXWS'`, `'RIDE_INIT'`) and `r` is a simple character vector representation of that variable's value.

4 The call_JSON_APL.c Sample

This sample demonstrating how to call APL functions using the JSON_APL Shared Object in C is built alongside the JSON_APL Shared Object. The source code is not included with standard Dyalog installations but is available from https://github.com/dyalog/NativeLib. It can be built using Visual Studio on Microsoft Windows or using `make` on Linux, macOS or on Windows using Cygwin. Details of how to build the sample are documented in *APL as a Shared Library*.

The examples below demonstrate how to use the entry points described in *Section 3.1*. The layout for each example is:

```
Function declaration (function arguments);

      Variable declaration and function call;
```

```
extern int LIBCALL Initialise(int runtime, unsigned int
len, wchar_t **args);

      const wchar_t *WSargs[] = {
          L"MAXWS=256Mb",
          L"SESSION_FILE=JSON_APL.dse"
      };
      Initialise(1, sizeof(WSargs) / sizeof(WSargs[0]),
      WSargs);
```

```
extern int LIBCALL CallJSON(wchar_t *function, wchar_t
*in, wchar_t *out, unsigned int len);

      // Load a .dyalog script
      APL = L"\"sign.dyalog\"";
      err = CallJSON(L"Load", APL, buf, 256);
      wprintf(L"CallJSON Load: %ls\nError: %i\n", buf,
      err);

      // Execute an APL function called using JSON
      APL =
      L"{\"Left\":[1,0,1,1,0],\"Statement\":\"/\",\"Right\
      ":\"APPLE\"}";
      err = CallJSON(L"Exec", APL, buf, 256);
      wprintf(L"CallJSON Exec: %ls\nError: %i\n", buf,
      err);
```

```
extern int LIBCALL ExecAPL(wchar_t *statement, wchar_t
*result);

      // Execute wchar APL statement (expects wide char
      return so format required):
      #define STR_FMT L"\x2355"
      wchar_t *APL = STR_FMT L"+/1 2 3 4";
      err = ExecAPL(APL, buf);
```

```
      wprintf(L"ExecAPL: %ls\n%ls\nError: %i\n", APL, buf,
      err);

extern int LIBCALL GetEnv(const wchar_t *name, wchar_t
*value, size_t len);

      // Query MAXWS
      err = GetEnv(L"MAXWS", buf, 256);
      wprintf(L"GetEnv MAXWS: %ls\nError: %i\n", buf,
      err);
```

5 The JSON_APL.py Sample

The Python example **JSON_APL.py** includes simple wrapper functions for accessing the shared library. These are:

`InitAPL(runtime, WSargs)`

Initialise the Dyalog interpreter with custom environment variables. The Boolean `runtime` tells the library whether to use the runtime interpreter (`1`) or development interpreter (`0`). `WSargs` is a list of Unicode strings that set environment variables, for example: `"MAXWS=512Mb"`.

If the Python example is suspended partway through execution (for example, by using an `input()` statement) then the active workspace can be accessed using the RIDE.

`CallJSON(function, parms)`

Call a function in the active workspace. This includes functions brought in using the APL function **`Load`** and those from **JSON_APL.dyalog** described in *Section 3.2*. The function **`Exec`** (in JSON_APL.dyalog) can process APL statements in JSON format. Either a whole statement can be passed as a single Unicode string (for example, `CallJSON("Exec","+⌿(⍳3)∘.+⍳5"))`, or a dictionary containing a `"Function"` value and, optionally, `"Right"` and `"Left"` values (corresponding to an APL function, right and left arguments respectively) can be passed. The dictionary construct allows Python numeric variables and lists to be passed as arguments to APL.

`CallJSON` returns a two-element list containing the function result and an error code (as described in *APL as a Shared Library*). It is often useful to retrieve numeric values in Python using `json.loads(result)`. **Note:** Only arrays of rank 1 or less can be passed using `CallJSON` (nested arrays are allowed).

`GetEnv(var)`

This is a wrapper for the APL function **`GetEnv`** described in *Section 3.2*.

The Python example uses the ctypes `create_unicode_buffer()` function to allocate mutable memory in which the shared library can store the results of function calls. Python is garbage-collected at the end of a run, but for languages that do not garbage-collect automatically the caller program is responsible for allocating and deallocating memory for the results of shared library calls.

The script detects the current platform (Microsoft Windows, macOS or Linux) and refers to the appropriate shared library and Conga paths given in **platformpaths.py**. To access the workspace using the RIDE, **platformpaths.py** must include the correct path to the Conga shared library.