The tool of thought for software solutions

APL as a Shared Library

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

Contents

1 About This Document

This document describes the API that can be used to call APL code in a shared library (that is, **.dll**, **.so** and **.dylib** files). It describes the SDK to create such a shared library and includes samples of how to call it.

1.1 Audience

It is assumed that the reader has a working knowledge of Dyalog (for information on the resources available to help develop your Dyalog knowledge, see http://www.dyalog.com/introduction.htm) as well as knowledge of the C programming language. Awareness of Dyalog's "Direct Workspace Access" SDK is also beneficial.

2 Introduction

On the Microsoft Windows operating system, it is possible for third-party code and applications to call APL code in shared libraries using the COM and .NET interfaces. However, these mechanisms are proprietary to Windows and are not available on all the operating systems supported by Dyalog. In addition, these interfaces impose restrictions and data formats that might not be optimal for use with APL code and the APL interpreter.

The SDK described in this document is implemented end-to-end with code and libraries (owned and implemented by Dyalog Ltd) that are efficient and portable across operating systems. They are also extensible, both by Dyalog Ltd and by end users.

The SDK comprises a number of C header files and libraries that can be used to produce a self-contained native shared library; this can then be called by any language or program that can access native shared libraries.

The SDK requires the author of the library to write some simple C source code to provide a strongly typed exportable interface to APL functions in a workspace or script.

On Microsoft Windows, shared libraries typically have a **.dll** extension.
On Linux and UNIX, shared libraries typically have a **.so** extension.
On macOS, shared libraries typically have a **.dylib** extension.

Although this SDK is intended to produce a *shared* library containing APL code (as described in this document), it is also possible to produce a *static* library or a self-contained application containing APL code.

3 Installation

3.1 Pre-requisites

To build and execute the samples, the host computer must have an installed C development environment.

On Microsoft Windows, Visual Studio 2015 or later is required to build and execute the sample VS solutions.

On Linux and macOS, the default C development and `make` tools should be sufficient to build and execute the samples. Additional developer libraries may be required along with those already installed on a developer machine, for example, the **ncurses** library. This can be installed with:

`sudo apt-get install libncurses5` (or equivalent)

3.2 Installation

The required native library files **dwa_static** and **libdyalog** are installed in **[DYALOG]/dwa/lib** as part of the standard installation of Dyalog.

The build process and necessary modifications to the build files for targeting specific platforms and versions of Dyalog are described in *Chapter 6*.

4 Architecture

4.1 The Binary

The SDK can be used to produce a library (shared or static) that contains both APL code and the Dyalog interpreter required to run the APL code.

The APL code, some interface C code and the Dyalog interpreter (as a static library) are all combined into a single, shared, binary.

On Microsoft Windows, the APL code is embedded in the binary as a Windows resource. On Linux and macOS, the APL code is embedded as static data.

4.1.1 The Interpreter

By default, the SDK refers to the absolute file paths of the native libraries **libdyalog** and **dwa_static** in a standard Dyalog installation. These files are in **[DYALOG]/dwa/lib**. *Section 6.2* describes how to alter the SDK to use a non-standard location.

For each shared library, the first time the `call_apl` function is called an embedded Dyalog interpreter is started and any APL code in the shared library is loaded into that interpreter. The interpreter instance persists for as long as the shared library is loaded into the host process. This makes it possible to persist data between calls in the APL workspace. By default, the interpreter is started as a runtime, meaning that a development environment is never presented. The interpreter is started with default values of MAXWS and other configuration parameters.

To tailor the properties of the APL interpreter, call the `load_apl` function before calling the `call_apl` function:

```
extern int load_apl(unsigned int flags, int argc, wchar_t
**argv);
```

The first argument to the load_apl function can be one of the following:

- 0 (zero) – on Microsoft Windows, any untrapped errors in the APL code will cause the Dyalog Development Environment to be displayed. On Linux/macOS, the interpreter will suspend until the RIDE is attached.
- ENGINE_F_RUNTIME – the interpreter behaves like a runtime system and will never display a development environment.

The `argv` argument to the `load_apl` function is an array of pointers to options (for example, "MAXWS=512Mb" or "-Dcw"). These parameters should be passed as wide character strings.

The `argc` argument to the `load_apl` function specifies how many entries there are in the `argv` array. The `load_apl` function starts the APL interpreter with the specified configuration.

If the appropriate settings are provided and the appropriate shared library files are installed with the shared library, then the Dyalog Development Environment (on Microsoft Windows) or the RIDE can be used to debug the APL code in the shared library.

4.1.1.1 Multithreading

Multiple threads can call APL code concurrently. This is achieved by mapping threads in the host application to multiple APL threads within the interpreter. If only a single thread is used to call APL code then it will run on APL thread one. APL thread zero is not used to run any APL code. For an example of multiple threads calling APL, see *Section 6.6.*

4.2 The Source

Each entry point in a shared library that will call APL code needs to be exported from the library. This is achieved by writing a C function that is declared as being callable from outside the library. (Note that in the SDK header files, this platform-specific mechanism is largely abstracted out). Each such function will typically contain a line of code for each of its parameters and for its result. This line tells the underlying library how to marshal the data from the native code to/from APL arrays.

For example, to call the APL function `inAPL` with a scalar integer argument:

```
extern int call_apl(const wchar_t *name,APL_PARAM *first,...);
EXPORT int LIBCALL inAPL(int month)
        {
        APL_INT32_PARAM(Month,month);
        return call_apl(L"inAPL",&Month,PARAM_END);
        }
```

In this example, the `APL_INT32_PARAM` statement (which is a macro from one of the SDK include files) declares that the `month` parameter to the function is to be passed as an `INT32` (int32_t) scalar to the APL code. The parameter is passed in a structure defined as a local variable called `Month`.

The `call_apl` function calls the APL function `inAPL`, passing the value of `month` (contained within the `Month` structure) as `inAPL`'s argument. The `call_apl` function takes a variable number of parameters; the special macro `PARAM_END` marks the end of the parameter list.

The result of the `call_apl` function is an error code:

- A return of 0 (zero) indicates that the call to the APL function (`inAPL` in this example) completed successfully.
- A non-zero return indicates that an error occurred during the invocation of the APL function (including marshalling the parameters, executing the APL code and marshalling the result):
    - Positive error codes have the same numeric value and meaning as APL error codes.
    - Negative error codes are reserved.

Parameters are passed in the right argument of the APL function (it is not possible to pass parameters in the left argument of an APL function).

In this example the APL function does not return a result. However, the return of `call_apl` can still indicate an error.

5 Implementation

5.1 Parameter and Result Macros

The SDK supports the following C data types:

- signed integers: 8, 16 and 32-bit
- floating point numbers: 32 and 64-bit
- unsigned characters: 8, 16 and 32-bit
- characters: char and wchar_t
- arrays of each of the above
- Null terminated arrays of the above characters
- arrays of the above characters containing JSON text

These types can be the input to, output of, or the result of an APL function.

C arrays will always be passed to the APL function as vectors.

Each of these parameters can be declared using a predefined macro. The names of these macros include the type that they are marshalling (see **call_apl.h** in any of the samples described in *Chapter 6* and *Appendix A* for the full list):

- `*_INT8_*` marshals `int8_t`
- `*_INT16_*` marshals `int16_t`
- `*_INT32_*` marshals `int32_t`
- `*_FLOAT_*` marshals `float`
- `*_DOUBLE_*` marshals `double`
- `*_CHAR_*` marshals `char` (as character(s))
- `*_CHAR8_*` marshals `uint8_t` (as character(s))
- `*_CHAR16_*` marshals `uint16_t` (as character(s))
- `*_CHAR32_*` marshals `uint32_t` (as character(s))
- `*_WCHAR_*` marshals `wchar_t` (as character(s))

These macros define local variables that encapsulate the value to be passed and other information that is necessary for the API to convert the data to an APL array. If it is not possible to marshal the specified value, then the `call_apl` function returns a non-zero error code.

A subset of the provided macros is described in the following sub-sections.

5.1.1 Scalar Parameters

`APL_INT32_PARAM(name,v)`

(also `APL_INT8_PARAM`, `APL_INT16_PARAM`, `APL_DOUBLE_PARAM`, and so on)

This macro defines a local variable called `name` which will marshal the value `v` (which is an `int32_t`) to the APL function.

`APL_CHAR8_PARAM(name,v)`

(also `APL_CHAR16_PARAM`, `APL_CHAR32_PARAM`, and so on)

This macro defines a local variable called `name` which will marshal the value `v` (which is an 8 bit unsigned char) to the APL function.

`APL_WCHAR_PARAM(name,v)`

This macro defines a local variable called `name` which will marshal the value `v` (which is a `whar_t`) to the APL function.

5.1.2 (Rank 1) Numeric Array Parameters

`APL_INT32_ARRAY_PARAM(name,v,l)`

(also `APL_INT8_ARRAY_PARAM`, `APL_INT16_ARRAY_PARAM`, and so on)

This macro defines a local variable called `name` which will marshal the array `v` (which is an `int32_t *`, of length `l`) to the APL function. The APL function will receive a vector of length `l`).

5.1.3 (Rank 1) Character Array Parameters

`APL_WCHAR_ARRAY_PARAM(name,v,l,f)`

This macro defines a local variable called `name` which will marshal the array `v` (which is a `wchar_t *`) to the APL function.

The argument `f` specifies additional information about the array `v`:

- If `f` includes the flag AP_NULLTERM, then `v` is null terminated, otherwise the length is provided by the parameter `l`.
- If `f` includes AP_JSON, then `s` is JSON text, in which case the API will deserialize the JSON (using ⎕JSON) before passing the array to the APL function:
  `APL_WCHAR_ARRAY_PARAM(var,param1,AP_NULLTERM|AP_JSON);`
  The above macro passes `param1`, (a null terminated JSON string) as a deserialised object (or array) to the APL function.

5.1.4 "Z" Format Data

`APL_ZFORMAT_PARAM(name,v)`

This macro defines a local variable called `name` which will marshal the binary data `v` (which of type `unsigned char*`) to the interpreter. The format of the data is the "Z" format of APL arrays as used by ⎕NA and TCPIP socket support. See the relevant documentation for more details of this format. Note that the length of the data is encoded within it so there is no length parameter to this macro.

5.1.5 Results

`APL_INT32_ARRAY_RESULT(name,v,l,f)` (and others)

This macro defines a local variable called `name` which will marshal the (vector) RESULT of the APL function to an array of `int32_t *`.

The argument `f` specifies additional information about how the array is allocated:

- If `f` includes `AP_SIZED` then `v` is a pre-allocated array, the size of which is specified by `l`.
- If `f` includes `AP_ALLOC` then the API will allocate space for the array before returning. It is the caller's responsibility to call `dwa_free()` on the array in `v` when the data is no longer required.

There are `_RESULT` versions of most of the macros previously described.

5.1.6 "Input and Output Parameters"

Sometimes it is convenient to use a single parameter as both INPUT to a function and as a placeholder for an additional "result" from a function. The `*_INOUT` macros provide this functionality. There are `*_INOUT` versions of the macros for all of the data types.

```
EXPORT void getsign_object(wchar_t *json,size_t len)
{
    APL_WCHAR_ARRAY_INOUT(JSON,json,len,AP_NULLTERM|AP_JSON|AP_SIZED);
    CHECK_ERR(call_apl(L"GetSignObject",&JSON,PARAM_END));
}
```

Here, the macro `APL_WCHAR_ARRAY_INOUT` is used to pass an `AP_NULLTERM` JSON string to the interpreter (the array will be deserialised before being passed to `GetSignObject`). The same macro specifies that the result of the function will be returned as an `AP_NULLTERM` JSON string. The use of the `AP_SIZED` macro will cause the function to fail if the RESULT array is larger than the specified size.

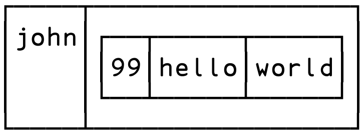
5.1.7 "Structures"

The macros `STRUCT_PARAM`, `STRUCT_INOUT` and `STRUCT_RESULT` can be used to create a nested array argument to an APL function. These macros combine multiple `APL_PARAM` structures into a single array.

For example:

```
typedef struct
         {
         int a;
         char *c8;
         wchar_t *wc;
         }test_str;


void test_struct(size_t len)
         {
         test_str str;
str.a=99;
str.c8="hello";
str.wc=L"world";

// extract str.a into A
APL_INT32_INOUT(A,str.a);
// extract str.c8 into C8
APL_CHAR8_ARRAY_INOUT(C8,str.c8,len,AP_SIZED|AP_NULLTERM);
// extract str.wc in WC
APL_WCHAR_ARRAY_INOUT(WC,str.wc,len,AP_SIZED|AP_NULLTERM);

// label as a separate parameter
char label[256]; strcpy(label,"John");
APL_CHAR8_ARRAY_INOUT(Label,label,256,AP_SIZED|AP_NULLTERM);

// combine a,c8 and wc into a nested array
APL_STRUCT_INOUT(Str,&A,&C8,&WC,PARAM_END);

// nested gets a 2 element vector
CHECK_ERR(call_apl(L"nested",&Label,&Str,PARAM_END));
}
```

The APL function `nested` will receive the following (shown with `]Box on`) as its argument:



```
┌────┬──────────────────┐
│john│┌──┬─────┬─────┐  │
│    ││99│hello│world│  │
│    │└──┴─────┴─────┘  │
└────┴──────────────────┘
```

5.1.8 Multiple "results"

If the argument list to `call_apl` includes multiple RESULT or OUT arguments, then the result of the APL function is distributed between these arguments (similar to stranded assignment). Other parameters are ignored for this purpose. If there is a length mismatch, then `call_apl` returns `5` (the APL value for LENGTH ERROR). Individual elements of the accumulated result are checked for type consistency.

5.2 Embedding APL Code

APL code can be embedded in the library as either APL script or as a workspace. The script/workspace file is specified in the **.rc** file as an RCDATA resource called either BOUND_RES_SCRIPT (for a script file) or BOUND_RES_DWS (for a workspace file).

```
BOUND_RES_SCRIPT RCDATA "sign.dyalog"
```

or

```
BOUND_RES_DWS RCDATA "sign.dws"
```

On Microsoft Windows, the **.rc** file is compiled using the Microsoft Resource Compiler. On Linux/macOS, a bash script is used to generate static data from the **.rc** file.

5.3 The Build System

The SDK is provided with a number of samples built with gmake and makefiles. These makefiles work on all operating systems. On Microsoft Windows there are also some Visual Studio 2015 solutions that can be modified to build your own libraries.

The default makefile (simply called **makefile**) can be used to build an APL/SO project from a single call file. As **makefile** is the default filename used by `make`, it is possible to build the APL/SO called `sign` with:

```
make THING=sign
```

This processes **sign.c**, **sign.dyalog** and **sign.rc** to produce **sign.dll**/**sign.so**/**sign.dylib** (on Microsoft Windows/Linux/macOS respectively).

If you have written **call<THING>.c** or **call<THING>cpp**, then

```
make -f makefile.callapl THING=sign
```

can be used to make **callsign** (**callsign.exe** on Microsoft Windows).

6 Samples

Dyalog provides a number of samples that use this technology. Each of these samples comprises source code, a Visual Studio 2015 solution file (for use on Microsoft Windows only) and a makefile (for use on other platforms, or on Windows with Cygwin installed). The samples each build two components; a shared library with embedded APL code, and an executable program that calls the shared library.

6.1 Installation

The source files for these samples – except for the interpreter native library files found in **[DYALOG]/dwa/lib** in a standard Dyalog installation – can be downloaded from https://github.com/dyalog/NativeLib and are provided under the MIT license. Use of the native library files is subject to the Dyalog license agreement.

6.2 Building

The default build process relies on the interpreter library files **dwa_static** and **libdyalog** being in the default location **[DYALOG]/dwa/lib** and could fail otherwise.

6.2.1 Building on Microsoft Windows

6.2.1.1 Building with Visual Studio

Open the chosen solution in VS 2015 and select **Build > Rebuild Solution** from the menu. The binaries for the solution are built in the typical Visual Studio output directories.

To change which directory the SDK looks in for **dwa_static** and **libdyalog**, select **Project > Properties** from the menu and in the **Linker > General** properties edit the **Additional Library Directories** to point to the desired location.

6.2.1.2 Building with make using Cygwin

Ensure **makefile** includes **makefile.win.<bits>** where **<bits>** is either 32 or 64, then follow the instructions for building on Linux and macOS.

6.2.2 Building on Linux and macOS

1. Edit line 2 in **makefile** to target a specific platform.

   ```
   include makefile.<platform>.<bits>
   ```

   where `<platform>` is one of `aix`, `linux`, `mac`, `pi` or `win` and `<bits>` is `64` (or `32` on Windows or AIX).

2. Modify line 3 to choose between debug and optimised builds.

```
MK_OPT:=$(if $(MK_OPT),$(MK_OPT),dbg)

MK_OPT:=$(if $(MK_OPT),$(MK_OPT),opt)
```

3. Build the sample:

```
> cd ~/NativeLib/<chosen sample>
> make
```

The make process builds the binaries in a subdirectory of **<chosen sample>/obj**.

To change which directory the SDK looks in for **dwa_static** and **libdyalog**, replace the path in **makefile.<platform>.<bits>** on the line beginning with `DWA_LIBDIR:=`.

6.3 Debugging

On all operating systems, C binaries can be debugged with the standard tools.

On Microsoft Windows, the APL code can be debugged by selecting the **Show Session** menu item from the Dyalog item in the system tray.

On Linux/macOS, the APL code can be debugged using the RIDE as long as RIDE_SERVE has been set appropriately. See the *RIDE User Guide* for more information.

6.4 Sample: HelloWorld

The **HelloWorld** sample calls into APL to display a simple "hello" message.

6.5 Sample: Sign

The **Sign** sample calls into APL using various different argument types to return the astrological star sign for a specific date.

6.6 Sample: Qa

The **Qa** sample uses and tests most (but not all) of the SDK and is provided for information only. It is used internally at Dyalog Ltd for QA purposes. Some elements of it might prove useful/interesting, for example, the **Qa** samples shows that the APL code can be called concurrently from multiple threads in the host application.

6.7 Sample: CallAPL

See *The JSON_APL Shared Object* document.

Appendix A Parameter and Result Macros

In the following definitions:

```
name:  The name of the local structure containing the value
first: The first parameter of a var_arg list in a nested definition
v:    The value to be passed
l: The length of an array (if required)
f: AP_* flags one or more of:
   AP_END  // used in PARAM_END to indicate the end of a parameter list
   AP_OUT  // a parameter which will contain a value after the call to call_apl
   AP_IN   // a parameter which will contain a value before the call to call_apl

   AP_NULLTERM  // a null terminated character buffer
   AP_JSON      // a JSON encoded character buffer
   AP_SIZED     // the array size will be specified in l
   AP_ALLOC     // the array will be allocated by call_apl (use dwa_free() to deallocate)
```

Many of these macros are based on other macro definitions so it is possible to extend this list with new definitions:

```
APL_STRUCT_PARAM(name,first,...) // an IN nested array / structure
APL_STRUCT_RESULT(name,first,...) // a RESULT nested array / structure

APL_STRUCT_INOUT(name,first,...) // an IN and OUT nested array / structure

APL_DOUBLE_PARAM(name,v)   // an 8 byte double IN parameter
APL_DOUBLE_RESULT(name)    // an 8 byte double RESULT

APL_INT8_PARAM(name,v) // a 1 byte SIGNED integer scalar for input
APL_INT8_INOUT(name,v) // a 1 byte SIGNED integer scalar for input and output
APL_INT8_RESULT(name) // a 1 byte SIGNED integer scalar for output
APL_INT16_PARAM(name,v) // a 2 byte SIGNED integer scalar for input
APL_INT16_INOUT(name,v) // a 2 byte SIGNED integer scalar for input and output
APL_INT16_RESULT(name) // a 2 byte SIGNED integer scalar for output
APL_INT32_PARAM(name,v) // a 4 byte SIGNED integer scalar for input
APL_INT32_INOUT(name,v) // a 4 byte SIGNED integer scalar for input and output
APL_INT32_RESULT(name) // a 4 byte SIGNED integer scalar for output

APL_CHAR_PARAM(name,v) // a 1 byte (default) character for input
APL_CHAR_INOUT(name,v) // a 1 byte (default) character for input and output
APL_CHAR_RESULT(name) // a 1 byte (default) character for output
APL_CHAR16_PARAM(name,v) // a 2 byte (unsigned) character for input
APL_CHAR16_INOUT(name,v) // a 2 byte (unsigned) character for input and output
APL_CHAR16_RESULT(name) // a 2 byte (unsigned) character for input
APL_WCHAR_PARAM(name,v) // a unicode character (width determined by OS) for input
APL_WCHAR_RESULT(name) // a unicode character (width determined by OS) for output

APL_INT8_ARRAY_PARAM(name,v,l) // a 1 byte SIGNED integer ARRAY for input
APL_INT8_ARRAY_INOUT(name,v,l,f) // a 1 byte SIGNED integer ARRAY for input and output
APL_INT8_ARRAY_RESULT(name,v,l,f) // a 1 byte SIGNED integer ARRAY for output
APL_INT16_ARRAY_PARAM(name,v,l) // a 2 byte SIGNED integer ARRAY for input
APL_INT16_ARRAY_INOUT(name,v,l,f) // a 2 byte SIGNED integer ARRAY for input and output
APL_INT16_ARRAY_RESULT(name,v,l,f) // a 2 byte SIGNED integer ARRAY for output
APL_INT32_ARRAY_PARAM(name,v,l) // a 4 byte SIGNED integer ARRAY for input
APL_INT32_ARRAY_INOUT(name,v,l,f) // a 4 byte SIGNED integer ARRAY for input and output
APL_INT32_ARRAY_RESULT(name,v,l,f) // a 4 byte SIGNED integer ARRAY for output

APL_INT64_ARRAY_INOUT(name,v,l,f) // a 8 byte SIGNED integer ARRAY for input and output
```

```
APL_INT64_ARRAY_RESULT(name,v,l,f) // a 8 byte SIGNED integer ARRAY for output

APL_DOUBLE_ARRAY_PARAM(name,v,l) // an 8 byte double ARRAY for input
APL_DOUBLE_ARRAY_INOUT(name,v,l,f) // an 8 byte double ARRAY for input and output
APL_DOUBLE_ARRAY_RESULT(name,v,l,f) // an 8 byte double ARRAY for output

APL_CHAR_ARRAY_PARAM(name,v,l,f)  // a 1 byte (default) character ARRAY for input
APL_CHAR_ARRAY_INOUT(name,v,l,f) // a 1 byte (default) character ARRAY for input and
output
APL_CHAR_ARRAY_RESULT(name,v,l,f) // a 1 byte (default) character ARRAY for output
APL_CHAR16_ARRAY_PARAM(name,v,l,f)
APL_CHAR16_ARRAY_RESULT(name,v,l,f)
APL_CHAR16_ARRAY_INOUT(name,v,l,f)
APL_CHAR32_ARRAY_PARAM(name,v,l,f)
APL_CHAR32_ARRAY_RESULT(name,v,l,f)
APL_CHAR32_ARRAY_INOUT(name,v,l,f)
APL_WCHAR_ARRAY_PARAM(name,v,l,f)
APL_WCHAR_ARRAY_RESULT(name,v,l,f)
APL_WCHAR_ARRAY_INOUT(name,v,l,f)
```