Wolfram Language Runtime (SDK) demo in Zig

Introduction

In this post I'll show a way of creating standalone applications via the Wolfram Language Runtime SDK (new in v14.1) using the Zig programming language (instead of C).
Why Zig? It's considered a modern successor to C, offering a transparent approach with no hidden control flow, memory allocations, preprocessor, or macros.
Plus, I have a personal preference for the language.

The full code lives in https://github.com/daneelsan/WolframLanguageRuntimeZigDemo/.

The Demo

The demo is an executable that uses the function `Transliterate` on an input.

C version:

This is the C code that creates such application:

```
#include <stdio.h>

#include "WolframLanguageRuntimeV1SDK.h"
```

```
int main(int argc, char *argv[]) {
    char *result;

    if (argc != 2) {
        printf("%s\n", "Usage: ./transliterate-c \"input\"");
        return 1;
    }

    // This is not in the blogpost, but is required for now to create an executable
    wlr_runtime_conf conf;
    wlr_InitializeRuntimeConfiguration(&conf);
    conf.containmentSetting = WLR_UNCONTAINED;

    wlr_err_t err0 = WLR_SDK_START_RUNTIME(WLR_EXECUTABLE,
WLR_LICENSE_OR_SIGNED_CODE_MODE, "/Applications/Wolfram.app/Contents", &conf);
        if (err0 != WLR_SUCCESS) {
        printf("%s (error: %d)\n", "SDK failed to start correctly", err0);
        return 1;
    }

    // Construct and evaluate the expression Transliterate[input]
    wlr_expr head = wlr_Symbol("Transliterate");
    wlr_expr arg = wlr_String(argv[1]);
    wlr_expr normal = wlr_E(head, arg);
    wlr_expr res = wlr_Eval(normal);

    // Extract the bytes from the string expression and print it
    wlr_err_t err = wlr_StringData(res, &result, NULL);
    if (err != WLR_SUCCESS) {
        return 1;
    }
    printf("%s\n", result);
    wlr_Release(result);

    return 0;
}
```

Zig version

Now I will show the equivalent Zig code:

```
const std = @import("std");
const wlr = @import("wlr.zig");

pub fn main() !void {
    // As I mentioned before, there is no hidden allocation so we have
    // to decide which allocator to use
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = gpa.allocator();

    var arg_iter = try std.process.argsWithAllocator(allocator);
    defer arg_iter.deinit();

    // Ignore the name of the executable
    _ = arg_iter.next().?;

    // If there is no argument to the executable, then print the usage
    const input = arg_iter.next() orelse {
```

```
        std.debug.print("Usage: ./transliterate-zig \"input\"\n", .{});
        return;
    };

    // Start the runtime
    try wlr.SDK.startRuntime(.{
        .app_type = .Executable,
        .license_mode = .LicenseOrSignedCode,
        .layout_dir = "/Applications/Wolfram.app/Contents",
        .containment_mode = .Uncontained,
    });

    // Construct and evaluate the expression Transliterate[input]
    const head = try wlr.Expr.symbol("Transliterate");
    const arg = try wlr.Expr.string(input);
    const normal = try head.construct(.{arg});
    const res = try normal.eval();

    // Extract the bytes from the string expression and print it
    const buffer = try res.stringData();
    defer wlr.release(buffer);
    std.debug.print("{s}\n", .{buffer});
}
```

Instead of using the C header file “WolframLanguageRuntimeV1SDK.h”, I manually wrote a wlr.zig file that performs the “same” functionality (I should mention that this file only has the required functionality to run the demo).
I could have gone with the route of using zig translate-c for automatic translation from the C-header, but some fringe C usages are not covered by yet it so I decided to against using it.

Code comparison

Let’s discuss and compare parts of the code

Memory Allocators

I mentioned that one of the reasons for choosing Zig over C is that there is no hidden memory allocations.
There is no default allocator in Zig, unlike C (**malloc**, **realloc**, etc.). Instead, functions which need to allocate accept an **Allocator** parameter (see Choosing an Allocator).

This can be seen in the following Zig code snippet:

```
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = gpa.allocator();

    var arg_iter = try std.process.argsWithAllocator(allocator);
    defer arg_iter.deinit();
```

Ideally we would use this same allocator and pass it to the Wolfram Language Runtime SDK so that it uses it for the allocations it needs.
This is not (yet?) possible so we will have to rely on the Wolfram Engine internal allocator.

Starting the Runtime

Let's see how the C version starts the runtime:

```
    wlr_runtime_conf conf;
    wlr_InitializeRuntimeConfiguration(&conf);
    conf.containmentSetting = WLR_UNCONTAINED;

    wlr_err_t err0 = WLR_SDK_START_RUNTIME(WLR_EXECUTABLE,
WLR_LICENSE_OR_SIGNED_CODE_MODE, "/Applications/Wolfram.app/Contents", &conf);
        if (err0 != WLR_SUCCESS) {
        printf("%s (error: %d)\n", "SDK failed to start correctly", err0);
        return 1;
    }
```

Now let's compare it to the Zig version:

```
    try wlr.SDK.startRuntime(.{
        .app_type = .Executable,
        .license_mode = .LicenseOrSignedCode,
        .layout_dir = "/Applications/Wolfram.app/Contents",
        .containment_mode = .Uncontained,
    });
```

In Zig, errors are values and may not be ignored. As such, any function that specifies that it could return an error (like **wlr.SDK.startRuntime**) needs to be handled explicitly (here I use try to handle it, which will simply propagate the error upwards).
This replaces the need to make the functions return an error as an integer, like C does.

Execute Wolfram Language code

Here is how the C version evaluates WL code:

```
    wlr_expr head = wlr_Symbol("Transliterate");
    wlr_expr arg = wlr_String(argv[1]);
    wlr_expr normal = wlr_E(head, arg);
    wlr_expr res = wlr_Eval(normal);
```

And here is how its done in Zig:

```
    const head = try wlr.Expr.symbol("Transliterate");
    const arg = try wlr.Expr.string(input);
    const normal = try head.construct(.{arg}); // .{ arg } is a tuple
    const res = try normal.eval();
```

There are two main differences:

- Since every function in the Zig version might return an error, then all calls have to handle this (again here I'm using try).
  This is why I am not able to write this Zig code as a one-liner, unlike the C version.

- Structs in Zig can have methods. They are not really special, they are only namespaced functions that you can call with dot syntax.

Extract the Result and Print it

Finally, extract the string from the String expression and print it in C:

```
    wlr_err_t err = wlr_StringData(res, &result, NULL);
    if (err != WLR_SUCCESS) {
        return 1;
    }
    printf("%s\n", result);
    wlr_Release(result);
```

In Zig this would look like:

```
    const buffer = try res.stringData();
    defer wlr.release(buffer);
    std.debug.print("{s}\n", .{buffer});
```

Unlike in the C version, the **stringData** method returns the string buffer. This is known as a slice in Zig, which is a pointer and a length.

Notice the defer statement, this will execute the release of the buffer at scope exit (in this case, at the end of the main function).
I already mentioned this in the “Memory Allocators” section but notice that I’m not passing the GPA allocator to the **release** function since it’s not supported to change the default allocator.

But how to build it?

C version

Zig comes with a C-compiler (see zig cc) and we will make use of it here:

```
$ zig cc main.c -o transliterate-c \

-L"/Applications/Wolfram.app/Contents/SystemFiles/Components/StandaloneApplicationsSD
K/MacOSX-x86-64/" \

-I"/Applications/Wolfram.app/Contents/SystemFiles/Components/StandaloneApplicationsSD
K/MacOSX-x86-64/" \
    -lstdc++ \
    -lStandaloneApplicationsSDK \
    -target x86_64-macos
```

Notice that even tough I’m compiling in ARM64 (see the Prerequisites section), I’m targeting the **x86_64-macos** architecture.
This is because there is a bug in the current MacOSX-ARM64 “StandaloneApplicationsSDK.a” library (though it should be fixed soon).

The executable will still be able to run because of Rosetta.

Now that the executable is compiled, see the usage:

```
$ ./transliterate-c
Usage: ./transliterate-c "input"
```

Use the executable:

```
$ ./transliterate-c 'しんばし'
shinbashi
```

Zig version

In this case, will use **zig build-exe** to create our executable:

```
$ zig build-exe main.zig --name transliterate-zig \

-L"/Applications/Wolfram.app/Contents/SystemFiles/Components/StandaloneApplicationsSD
K/MacOSX-x86-64/" \
    -lStandaloneApplicationsSDK \
    -lc++ \
    -target x86_64-macos
```

Now that the executable is compiled, see the usage:

```
$ ./transliterate-zig
Usage: ./transliterate-zig "input"
```

The executable works the same as the C executable:

```
$ ./transliterate-zig 'しんばし'
shinbashi
```

Conclusions

In this demonstration, I successfully illustrated how to create a standalone application using the Wolfram Language Runtime SDK with the Zig programming language, as an alternative to C.
By manually writing a wlr.zig file, I was able to replicate the functionality provided by the C header file WolframLanguageRuntimeV1SDK.h.
Although some might say Zig presents a steeper learning curve due to its strict error handling and lack of implicit memory allocation, it offers clearer and more predictable code execution.
Ultimately, the Zig version of the application demonstrated here achieves the same results as the C version, with the added benefits of Zig's modern features and safer programming practices.

References

- Wolfram Language Runtime (SDK) demo in Zig
- Yet More New Ideas and New Functions: Launching Version 14.1 of Wolfram Language & Mathematica - Standalone Wolfram Language Applications!
- Zig Language Reference
- Zig Standard Library

Prerequisites

Wolfram Language Version

I mentioned this already, but you'll need a Wolfram Engine version >14.1.

In[3]:=
```
$VersionNumber
```

Out[3]=
```
14.1
```

Another thing to make sure is to rename the SDK library found in /Applications/Wolfram.app/Contents/SystemFiles/Components/StandaloneApplicationsSDK/MacOSX-x86-64/ to have the **lib** prefix.
This is because libraries are expected to have them in Unix-like systems.
Otherwise, **zig cc** and **zig build-exe** won't be able to find the library when using **-lStandaloneApplicationsSDK:**

```
Module[{dir = FileNameJoin[{$InstallationDirectory,
      "/SystemFiles/Components/StandaloneApplicationsSDK/MacOSX-x86-64/"}]},
 CopyFile[
  FileNameJoin[{dir, "StandaloneApplicationsSDK.a"}],
  FileNameJoin[{dir, "lib" <> "StandaloneApplicationsSDK.a"}],
  OverwriteTarget → True
 ]
]
```

Out[22]=
```
/Applications/Wolfram14.1.app/Contents/SystemFiles/Components/
  StandaloneApplicationsSDK/MacOSX-x86-64/libStandaloneApplicationsSDK.a
```

Platform

I've tested this (solely) on my M1 mac:

In[11]:=
```
$System
```

Out[11]=
```
Mac OS X ARM (64-bit)
```

Zig Version

I used the following Zig version:

```
$ zig version
0.14.0-dev.823+624fa8523
```

I should mention Zig is not on version v1 .0 yet.
The language is in constant development and some things might (and will) break in the future.