Embedding Webassembly in a FPC program

Michaël Van Canneyt

November 3, 2023

Abstract

Webassembly was designed to run in the browser. It's design is focused on simplicity and safety, making it ideal for sandboxing. As a result more and more it finds its way in applications that run outside the browser. In this article we show how to embed a Webassembly Module in a Free Pascal module.

1 Introduction

WebAssembly is a an open bytecode format similar in purpose to the Java and C# bytecode formats: it is designed to run in a sandboxed environment. The initial target of this was the browser, where it allows computationally intensive tasks to be run in the browser at speeds that are vastly superior to the speed of plain javascript.

Today, you can compile from any programming language (Notably C, C++, Rust and of course Pascal) to the webassembly format: the LLVM compiler supports WebAssembly as an output format.

The specification of this bytecode format is open and managed by the W3C consortium:

`https://www.w3.org/TR/wasm-core-2/`

The specification is maintained on github:

`https://github.com/WebAssembly/design`

Beside the core specification, which describes the basic bytecode format and the supported assembly instructions, there are also various extensions. A list of extensions and their various stages of implementation can be found on github as well:

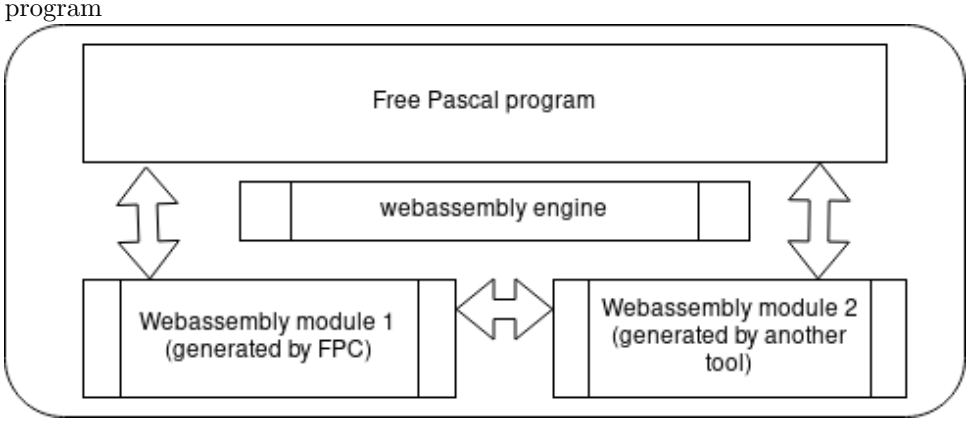
`https://github.com/WebAssembly/proposals`

Some of the more interesting ones are threading and exception support.

The open character of the format means anyone can implement a runtime that loads and execute the format. In fact, outside the browser, several wasm execution environments exist:

WasmTime This engine is maintained by the Bytecode Alliance, which actively supports the development of WebAssembly. It is a conservative implementation, meaning that it only supports established proposals of the WebAssembly specification.

Figure 1: Using a webassembly engine to run a WebAssembly program in a FPC program



`https://wasmtime.dev/`

WasmEdge This engine is maintained by an independent community of developers and was recently brought under the umbrella of the Cloud native computing foundation https://cncf.io/, itself part of the Linux foundation. This implementation is more cutting edge , it supports many of the more experimental WebAssembly extensions.

`https://wasmedge.org/`

Wasmer is an independent implementation of an webassembly bytecode engine. Like wasmedge, it is more accepting of new proposals:

`https://wasmer.io/`

It has adopted an approach similar to npm (the node package manager): it has a package system with ready-to-run webassembly modules.

WAMR is another bytecode alliance implementation of the webassembly runtime, focused on small memory footprint and fast execution:

`https://github.com/bytecodealliance/wasm-micro-runtime`

All these implementations have a library which you can use to embed the engine in your application: this means you can run a webassembly program (which can consist of multiple webassembly modules linked together) embedded in your Free Pascal program.

This embedded webassembly program can also be generated by Free Pascal or by some other programming tool or - most likely - a combination of both: All engines support linking together various webassembly modules, regardless of the language they were originally programmed in. The format used by webassembly ensures that all modules use the same format to exchange data and code.

This is shown diagrammatically in figure 1 on page 2: a native Free Pascal host program loads 2 webassembly modules: one written in Free Pascal and one written in another language, and can execute functions in both modules. The modules themselves can also call functions in each other.

Of these, the wasmtime and wasmedge engines have at least some comprehensive documentation, and therefore import units for these libraries have been created for Free Pascal, which we will demonstrate here.

2 The WASI specification

The WebAssembly specification by itself does not specify how to interact with the environment: The format does not describe how to read and write files, get the time and so on.

It only describes a mechanism how to import functions from the runtime environment. Obviously, it also specifies how to execute functions in the webassembly. This allows for modules to be chained together, just as dynamically loadable libraries.

Naturally, a bytecode format that cannot interact with the environment is of little use. Therefore a separate specification was developed which provides a minimal list of functions needed to interact with the outside world: WASI: the WebAssembly System Interface.

`https://github.com/WebAssembly/WASI`

All engines specified above support this interface. This means that when a webassembly module is loaded into one of these engines, the functions listed in WASI are available. It serves as the basis for a LibC implementation that runs in a webassembly environment.

The WASI current specification has only basic OS interaction support: only basic file I/O and getting the time and environment variables. That means no graphical environment, no TCP/IP or HTTP environment etc. (The latter are however expected to appear in version 2 of the spec)

In essence, the spec provides enough calls to implement the SysUtils unit in Free Pascal, and in essence that is what has been used to develop the Free Pascal WebAssembly target.

Putting all this together, it means basic Free Pascal programs can be loaded into one of the engines mentioned above.

Does this mean you cannot run more advanced code (requiring sockets, UI etc.) in these runtimes? No, of course not: the engines support providing your own functions to the webassembly. That means that if you provide functions to execute a HTTP request, then these functions can be executed from inside the runtime to download HTML pages. It should be noted that you must be careful with the functionality you provide to the webassembly: functions open doors to the environment which can potentially be exploited.

3 Using wasmtime

Wasmtime is available as a command-line tool with which you can start a webassembly module from the command-line. This command-line tool itself is simply a shell around the wasmtime dynamically loadable library.

Instructions for downloading and installing wasmtime can be found here

`https://docs.wasmtime.dev/cli-install.html`

Binaries of the releases for all major platforms can be found here:

`https://github.com/bytecodealliance/wasmtime/releases`

Free pascal contains a unit calledwasmtime which can be used to access the functionality of the wasmtime library. The library is loaded at runtime with the `LoadWasmTime` call:

```
Procedure LoadWasmTime(const Lib : string);
```

The `lib` argument is the name of the library file to load. The name of the library as distributed is available in the `libwasmtime` constant. If the loading fails, an exception is raised.

The library exposes well over 100 types and 500 functions, this is clearly more than can be explained in the context of a single article. Therefore we'll describe a simple example which demonstrates how to load a library, make a host-provided function available to the wasm module, and show how this function is called.

The webassembly program which we will be loading and executing is quite simple:

```
(module
  (func $hello (import "" "hello"))
  (func (export "run") (call $hello))
)
```

Without going into the details of the webassembly text format, it is apparent from the text that this module imports a function called `hello` (with no parameters and no return value) and exports a function called `run` which simply calls the imported "hello" function. The `run` function again has no parameters and return value.

Note that no file IO or other external functions are used: just one imported function and one exported function. There is also no initialization or finalization code.

To execute this webassembly program, we need therefore to load this file, convert it to bytecode, provide it with a `hello` function and then call the `run` function. The expectation is that the `hello` function is called, and that the `run` function returns immediatly afterwards.

The main program starts by declaring a lot of variables:

```
Var
  engine : Pwasm_engine_t = Nil;
  store : Pwasmtime_store_t = Nil;
  context : Pwasmtime_context_t = Nil;
  F : TMemoryStream;
  wat : Twasm_byte_vec_t;
  wasm : twasm_byte_vec_t;
  module : Pwasmtime_module_t = Nil;
  error : Pwasmtime_error_t = Nil;
  hello_ty : Pwasm_functype_t = nil;
  hello : Twasmtime_func_t;
  trap : Pwasm_trap_t = Nil;
  instance : Twasmtime_instance_t;
  import : Twasmtime_extern_t;
  run : Twasmtime_extern_t;
  ok : Byte;
```

The meaning of these variables will be explained as we encounter them in the program code.

All types used in WasmTime are opaque record types: the exact details of the record are not exposed. Most of these records are created dynamically with a functions that returns a pointer to such an opaque type, and as a rule the function name ends in (or contains) `_new`. When you are done with a particular variable, you must release the memory occupied by the variable using a function whose name ends in `_delete`.

The program of cource starts by loading the wasmtime library. When this has succeeded, a webassembly engine is created using the `wasm_engine_new` function.

```
begin
  Writeln('Loading wasm library');
  Loadwasmtime('./'+libwasmtime);
  Writeln('Initializing...');
  engine := wasm_engine_new();
  store:=wasmtime_store_new(engine, nil,nil);
  context:=wasmtime_store_context(store);
```

The store is a general purpose memory area for the engine. It can be used to add user data, but is also used by the engine. The context is is a pointer used by the engine to add/remove data to the store.

The following piece of code will load a file containing a webassembly module using text representation (a kind of assembly language). It allocates a memory area (`wat`) using the `Twasm_byte_vec_t` type (which represents a memory block) needed by the engine, and moves the contents of the file into it:

```
F:=TMemoryStream.Create;
try
  F.LoadFromFile('hello.wat');
  wasm_byte_vec_new_uninitialized(@wat, F.Size);
  Move(F.Memory^,wat.data^,F.Size);
finally
  F.Free;
end;
```

In the following step, the text representation of the webassembly module is converted to bytecode using `wasmtime_wat2wasm` and stored in a memory block `wasm`. (again of type `twasm_byte_vec_t`. The text representation of the module (`wat`) is disposed of.

```
Writeln('Compiling module...');
error:=wasmtime_wat2wasm(PAnsiChar(wat.data), wat.size, @wasm);
if (error<>Nil) then
  exit_with_error('failed to parse wat', error, Nil);
wasm_byte_vec_delete(@wat);

error:=wasmtime_module_new(engine, Puint8_t(wasm.data), wasm.size, @module);
wasm_byte_vec_delete(@wasm);
if (error <> nil) then
  exit_with_error('failed to compile module', error, nil);
```

After compiling the webassembly, the bytecode is loaded into a module (`module`, of type `Pwasmtime_module_t`, with the `wasmtime_module_new` function, and the bytecode representation is discarded. The module is what will be used when executing the webassembly.

The `exit_with_error` function is an auxiliary function which will be used in several locations in the program. We'll come back to it later.

At this point we have a module, ready to be executed. We did not yet use the context which we created at the beginning. Now we get to the point where this context will be used: We will provide a pascal 'hello' function to the webassembly module.

Functions provided to the webassembly module must have the appropriate function type:

```
Twasmtime_func_callback_t = function (env:pointer;
                                      caller:Pwasmtime_caller_t;
                                      args:Pwasmtime_val_t;
                                      nargs:Tsize_t;
                                      results:Pwasmtime_val_t;
                                      nresults:Tsize_t):Pwasm_trap_t;cdecl;
```

The `env` argument can be used to pass information along, for example the `Self` pointer of an object. The `caller` contains information about the calling environment, and the `args` pointer points to the arguments passed during the call. The `nargs` parameter contains the number of arguments. Similarly, the `results` and `nresults` arguments are used to specify return values.

The return value is a `trap` (of type `Pwasm_trap_t`): when non-nil, it signals an error condition to the webassembly engine.

Knowing this, our 'Hello' function looks like this:

```
function hello_callback(env : Pointer;
                        caller : Pwasmtime_caller_t;
                        args : pwasmtime_val_t;
                        nargs : size_t;
                        results : pwasmtime_val_t; nresults : size_t) : pwasm_trap
begin
  Writeln('Calling back...');
  Writeln(' Hello World!');
  Result:=Nil;
end;
```

The next part of our program is defining this function in the webassembly module, so it can be called. This starts by creating a function type (`hello_ty`, of type `Pwasm_functype_t`), which is then registered as a function using `wasmtime_func_new`.

A function type is represented by `Pwasm_functype_t`. This corresponds to a procedural type in pascal. The WebAssembly format defines a function type for all functions and procedures: For both internal and external functions, a function type must be defined. For external (imported/exported) functions, this is logical: the runtime engine needs to know what data to provide or what date to extract whenever the boundary between webassembly and the host environment is crossed: both when calling a webassembly function in a webassembly module and when an external function is called by the webassembly module.

To register a callable function, the `wasmtime_func_new` is used:

```
 procedure wasmtime_func_new (
      store: Pwasmtime_context_t;
      _type: Pwasm_functype_t;
      callback: Twasmtime_func_callback_t;
      env: pointer;
      finalizer: TFinalizer;
      ret: Pwasmtime_func_t)
```

The second argument (`_type`) is the function type, and the third (`callback`) is the actual function to call.

The `env` argument can be filled with anything you like, it will be passed as-is when the callback is called. This can be used for example to store an object pointer. Finally a `finalizer` for `Env` can be specified, this is a function which is called typically to free the env object when the webassembly module is destroyed. The `ret` argument points to a location which will be filled with a function definition.

Since our function accepts no arguments and returns no results, the function type and the registration of the callback is quite simple:

```
Writeln('Creating callback...\n');
hello_ty:=wasm_functype_new_0_0();
wasmtime_func_new(context, hello_ty, @hello_callback, Nil, Nil, @hello);
```

Note the `context` argument and the `hello` variable which will contain the function definition as created by the wasm runtime.

What we did till now is define webassembly module, and the functions which we will be providing to it. It is ready to run.

To actually run a webassembly, we must create an instance of the module (it is possible to create multiple instances of a single module, and execute them in parallel). From this instance we can then extract the address of the exported function ('run'), and call it.

To create an instance of a webassembly module, the `wasmtime_instance_new` function is used.

```
function wasmtime_instance_new(
      store: Pwasmtime_context_t;
      module: Pwasmtime_module_t;
      imports: Pwasmtime_extern_t;
      nimports: Tsize_t;
      instance: Pwasmtime_instance_t;
      trap: PPwasm_trap_t):Pwasmtime_error_t
```

The `store` is the context we are using, the `module` is the module we just defined. As can be seen from the `imports` argument, we must provide it with all the functions that can be used. If several instances can be created and run, it makes sense that the functions are provided to the instance, and not to the module: the `env` pointer for the callable functions will typically be different for each instance. Upon successful return, `instance` will be filled with a runnable instance. `trap` will be filled with an error report if an error occurred.

Errors can happen for example when the module expects to be able to import functions `foo` and `bar`, but only `bar` is supplied.

In our case, we need to provide the `hello` function we just created:

```
Writeln('Instantiating module...');
import.kind:=WASMTIME_EXTERN_FUNC;
import.of_.func:=hello;
error:=wasmtime_instance_new(context, module, @import, 1, @instance, @trap);
if (error<>nil) or (trap <>Nil) then
  exit_with_error('failed to instantiate', error, trap);
```

The arguments to the `wasmtime_instance_new` function are the context, the module, 1 import definition, and 2 variables for return values `error` and `trap`, which we examine on return.

The function that is exported from the webassembly module is called 'run'. We extract the function definition from the instance using the `wasmtime_instance_export_get` function:

```
function wasmtime_instance_export_get(
     store: Pwasmtime_context_t;
     instance: Pwasmtime_instance_t;
     name: PAnsiChar;
     name_len: Tsize_t;
     item: Pwasmtime_extern_t):T_Bool;
```

The `store` and `instance` arguments are of course the context we are using and the instance we just created. The `name` and `name_len` functions are used to pass the name of the function you wish to have ('run' in our case) and the `item` is filled with the function definition on return: when the function exists, the function returns a nonzero return value.

So, our code to get the 'run' function definition is:

```
Writeln('Extracting export...\n');
ok:=wasmtime_instance_export_get(context, @instance, PAnsiChar('run'), 3, @run) ;
if OK=0 then
  exit_with_error('failed to get run export', nil, nil);
if run.kind<>WASMTIME_EXTERN_FUNC then
  exit_with_error('run is not a function', nil, nil);
```

The `run` variable holds a reference to the exported function.

Now we are ready to actually run the function. This is done with the `wasmtime_func_call`:

```
  function wasmtime_func_call (
      store: Pwasmtime_context_t;
      func: Pwasmtime_func_t;
      args: Pwasmtime_val_t;
      nargs: Tsize_t;
      results: Pwasmtime_val_t;
      nresults: Tsize_t;
      trap: PPwasm_trap_t): Pwasmtime_error_t;
```

The first 2 arguments are the `store` context and the function definition we just extracted. Note that the wasm module or instance do not need to be specified: they are implicit in the function definition. The arguments to be provided to the called function and results returned by it, are specified in the next 4 arguments. The last argument (`trap`) is used to hold an error condition when something goes wrong.

Since the 'run' function does not take arguments, and provides no results, we do not need to set up anything to specify them, so we are ready to call our function:

```
Writeln('Calling export...');
error:=wasmtime_func_call(context, @run.of_.func, nil, 0, nil, 0, @trap);
if (error<>nil) or (trap<>nil) then
  exit_with_error('failed to call function', error, trap);
```

The first thing to do is to check if an error was returned.

After all this, the 'run' function has been called, and the instance and module can be cleaned up. To clean up, we clean up the module and the `store` context: everything connected to the store will also be cleaned up:

```
  Writeln('All finished!\n');
  wasmtime_module_delete(module);
  wasmtime_store_delete(store);
  wasm_engine_delete(engine);
end.
```

All that remains to be done is to show the `exit_with_error` procedure: This procedure shows the error information returned by the wasmtime engine, and demonstrates that you must release the `trap` and `error` runtime error reports when they occur. Failure to do so will result in memory leaks:

```
procedure exit_with_error(message : PAnsiChar; error : Pwasmtime_error_t; trap: Pwasm_trap_t)

var
  error_message : Twasm_byte_vec_t ;
  S : AnsiString;

begin
  Writeln(stderr, 'error: ', message);
  S:='';
  if (error <> Nil)  then
    begin
    wasmtime_error_message(error, @error_message);
    wasmtime_error_delete(error)
    end
  else
    begin
    wasm_trap_message(trap, @error_message);
    wasm_trap_delete(trap);
    end;
  SetLength(S,error_message.size);
  Move(error_message.data^,S[1],error_message.size);
  Writeln(stderr, S);
  wasm_byte_vec_delete(@error_message);
  halt(1);
end;
```

With all this in place, we can now run the binary. If all goes well, you can see output similar to the one shown in figure 2 on page 10.

4 Providing a WASI environment to execute a FPC-generated program

The previous demonstration program only used an imported function (hello) and an exported function (run) to communicate with the outside world. In particular, it did not use any WASI functionality. The webassembly RTL of Free Pascal does use the WASI functionality. `wasmtime` does not make the WASI interface available to a webassembly module unless you instruct it to. Since the FPC RTL for webassembly

Figure 2: The first wasmtime example program in action

relies on the WASI interface, we'll execute a FPC-generated program to demonstrate how to provide the WASI functionality to a webassembly module.

The FPC program is a very simple 'Hello, world' :

```
begin
  Writeln('"Hello, World!" from FPC webassembly');
end.
```

When the Free Pascal webassembly compiler and RTL are installed, then compiling this program can be done so:

```
ppcrosswasm32 hello.pp
```

If all went well, it results in a `hello.wasm` webassembly module.

The following Free Pascal program will load the webassembly module and provide the WASI environment. The variable declaration block closely resembles the one in the previous example, we only list the additional variables that were not present in the previous program:

```
var
  linker : Pwasmtime_linker_t;
  wasi_config : Pwasi_config_t;

begin
  Writeln('Loading wasm library');
  Loadwasmtime('./'+libwasmtime);
  Writeln('Initializing...');
  engine := wasm_engine_new();
  store:=wasmtime_store_new(engine, nil,nil);
  context:=wasmtime_store_context(store);
```

```
linker:= wasmtime_linker_new(engine);
error:=wasmtime_linker_define_wasi(linker);
if (error<>Nil) then
  exit_with_error('failed to define link wasi', error, Nil);
```

Here we create a webassembly linker, and use it to link the WASI functionality to our webassembly module: the `wasmtime_linker_define_wasi` function makes the WASI standard functions availabe in the webassembly module. However, the WASI functions needs to be configured: what filesystem directories are available, what are the environment variables, command-line parameters ? What to do with standard input, output and error output file descriptors ?

All this can be specified by creating a WASI configuration, using the `wasi_config_new` function:

```
wasi_config:=wasi_config_new();
if (wasi_config=nil) then
  exit_with_error('failed to create wasi config', Nil, nil);
```

The wasi configuration must be configured with one or more configuration functions:

wasi_config_set_argv Sets values for the command-line parameters of the wasm module.

wasi_config_inherit_argv Uses the values of the host program for the command-line parameters of the wasm module.

wasi_config_set_env Sets the values for the environment variables of the wasm module.

wasi_config_inherit_env Uses the values of the host program environment variables for the wasm module.

wasi_config_set_stdin_file Specifies a file to be used as standard input for the webassembly program.

wasi_config_set_stdin_bytes Specifies a memory block to be used as standard input for the webassembly program.

wasi_config_inherit_stdin Sets the standard input of the host program as standard input for the webassembly program.

wasi_config_set_stdout_file Specifies a file to be used as standard output for the webassembly program.

wasi_config_inherit_stdout Sets the standard output of the host program as standard output for the webassembly program.

wasi_config_set_stderr_file Specifies a file to be used as standard error output for the webassembly program.

wasi_config_inherit_stderr Sets the standard erro output of the host program as standard erroroutput for the webassembly program.

wasi_config_preopen_dir Configures a "preopened directory" as base directory for WASI file APIs.

For our simple demonstration, we'll just inherit everything from the host environment, and make the current directory available:

```
wasi_config_inherit_argv(wasi_config);
wasi_config_inherit_env(wasi_config);
wasi_config_inherit_stdin(wasi_config);
wasi_config_inherit_stdout(wasi_config);
wasi_config_inherit_stderr(wasi_config);
wasi_config_preopen_dir(wasi_config,PAnsiChar('.'),PAnsiChar('.'));
error:=wasmtime_context_set_wasi(context, wasi_config);
if (error<>Nil) then
  exit_with_error('failed to instantiate WASI', error, nil);
```

The `wasmtime_context_set_wasi` function couples the WASI configuration to the WASI environment of the webassembly module.

We can now load the module and execute it.

Loading the webassembly module differs somewhat from our previous program: instead of loading a webassembly text format and compiling it, we're loading an already compiled `.wasm` module:

```
F:=TMemoryStream.Create;
try
  F.LoadFromFile('hello.wasm');
  wasm_byte_vec_new_uninitialized(@wasm, F.Size);
  Move(F.Memory^,wasm.data^,F.Size);
finally
  F.Free;
end;

// Now that we've got our binary webassembly we can create our module.
Writeln('Creating module...');
error:=wasmtime_module_new(engine, Puint8_t(wasm.data), wasm.size, @module);
wasm_byte_vec_delete(@wasm);
if (error <> nil) then
  exit_with_error('failed to compile module', error, nil);
```

This time we use the webassembly linker to instantiate the module, as the linker needs to provide the WASI functionality to the webassembly module. The function to do so is called `wasmtime_linker_module`:

```
function wasmtime_linker_module(
      linker: Pwasmtime_linker_t;
      store:Pwasmtime_context_t;
      name:PAnsiChar;
      name_len:Tsize_t;
      module:Pwasmtime_module_t): Pwasmtime_error_t;
```

The name of the module can be specified in the `name` and `name_len` variables. We don't use them here: they are only needed when various modules must be linked together, because the linker will link imports from one module to exports of another module using the module name.

Since we're loading only one module, it is not necessary to specify a name:

```
// Instantiate the module
error:=wasmtime_linker_module(linker, context, Nil, 0, module);
if (error<>nil) then
  exit_with_error('failed to instantiate module', Nil, Nil);
```

A module can have a default exported function: This is the '_start' symbol which starts the Free Pascal program. We extract the value of this function using `wasmtime_linker_get_default`, and call it to start the FPC generated program:

```
error:=wasmtime_linker_get_default(linker, context, nil, 0, @func);
if (error<>nil) then
   exit_with_error('failed to locate default export for module', error, nil);
// And call it!
Writeln('Calling export...');
error:=wasmtime_func_call(context, @func, nil, 0, nil, 0, @trap);
if wasmtime_error_exit_status(error,@status)<>0 then
  Writeln('Wasm program exited with status: ',Status)
else
  exit_with_error('Error while running default export for module', error, trap);
```

The `exit_proc` routine in the WASI specifition exits the Webassembly program. The Free Pascal runtime for webassembly calls this when the program is halted. In `wasmtime`, the `exit_proc` routine raises an error to halt the program, so strangely enough, the result of running the start function is an error condition! Luckily, the `wasmtime_error_exit_status` function can be used to check for this special case.

When the program has exited, all that is left to do is to clean up, just as in the previous example program:

```
// Clean up after ourselves at this point
Writeln('All finished!\n');
wasmtime_module_delete(module);
wasmtime_store_delete(store);
wasm_engine_delete(engine);
end.
```

The result of this can be seen in figure 3 on page 14.

5 Using WasmEdge

A second library that can be used to embed WebAssembly programs is `wasmedge`. Installation instructions can be found on

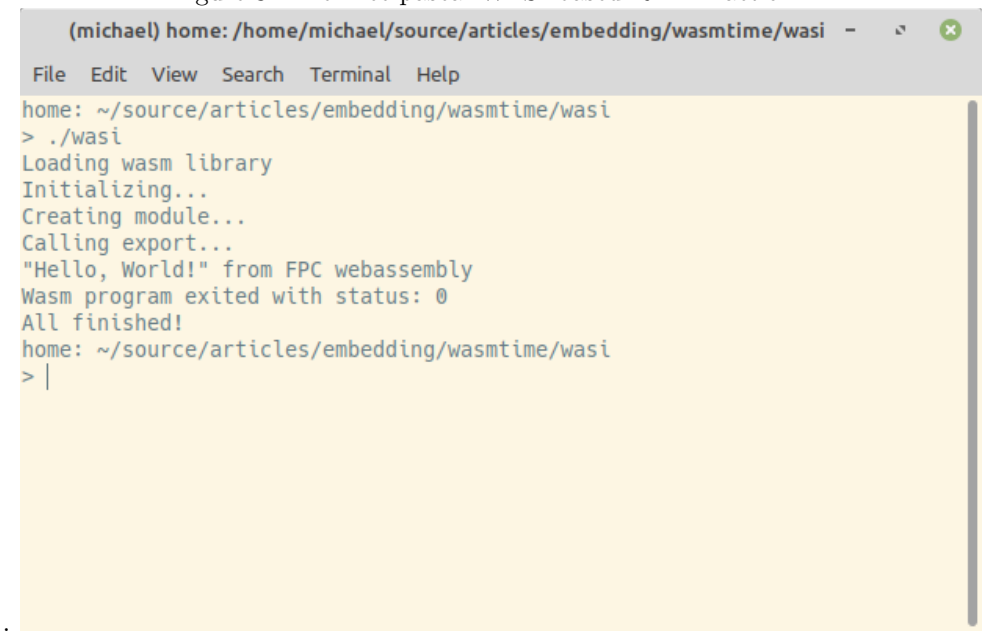
`https://wasmedge.org/docs/start/install/`

The unit that imports this library is called `libwasmedge`. The library works largely similar to the `wasmtime` library, but differs in the details. In some ways it is simpler than the `wasmtime` library. It only exposes 300 functions - still a considerable number, but less than the wasmtime library.

The sample program we will demonstrate loads and runs the following webassembly function which calculates the fibonacci series:

```
(module
```

Figure 3: The Free pascal WASI-based RTL in action



.

```
  (func $fib (export "fib") (param $n i32) (result i32)
    local.get $n
    i32.const 2
    i32.lt_s
    if
      i32.const 1
      return
    end
    local.get $n
    i32.const 2
    i32.sub
    call $fib
    local.get $n
    i32.const 1
    i32.sub
    call $fib
    i32.add
    return
  )
)
```

Without going into the details of the webassembly format, you can see in the second line that it defines a function 'fib' which accepts a 32 bit integer as a parameter and which returns another 32-bit integer.

The program to call this function is relatively simple:

```
uses ctypes, libwasmedge;

var
  ConfCxt : PWasmEdge_ConfigureContext;
  VMCxt : PWasmEdge_VMContext;
```

```
  Returns, Params : Array[0..0] of TWasmEdge_Value;
  FuncName : TWasmEdge_String;
  Res : TWasmEdge_Result;
  pmodule : pcchar;

begin
  Writeln('Loading library...');
  Loadlibwasmedge('./'+libwasmname);
  ConfCxt:=WasmEdge_ConfigureCreate();

  Writeln('Adding WASI environment...');
  WasmEdge_ConfigureAddHostRegistration(ConfCxt, WasmEdge_HostRegistration_Wasi);
  Writeln('Creating engine...');
  VMCxt:=WasmEdge_VMCreate(ConfCxt,Nil);
```

After loading the library, a configuration context is created using `WasmEdge_ConfigureCreate`. The `WASI` environment is added to the configuration using the `WasmEdge_ConfigureAddHostRegistration` routine.

Within this context, a 'virtual machine' is created that will execute the webassembly module. The `WasmEdge_VMCreate` function is used to create this virtual machine:

```
function WasmEdge_VMCreate(
          ConfCxt: PWasmEdge_ConfigureContext;
          StoreCxt: PWasmEdge_StoreContext):PWasmEdge_VMContext;
```

The parameters are the configuration and a store (similar to what is used in wasmtime). The store is not needed for this example.

To call the fibonacci function, a parameter is needed. This parameter is generated by the `WasmEdge_ValueGenI32` function. Webassembly knows only a few basic types (integer, float) so the number of functions that you must use to create values is limited: there are only 8 functions of which you will use 4 in practice.

```
  Params[0] := WasmEdge_ValueGenI32(32);
  FuncName:=WasmEdge_StringCreateByCString(Pcchar(Pansichar('fib')));
```

The second line creates a string 'fib' that can be used by the wasmedge library. This string is used to call the fibonacci. Calling a function in a webassembly module can be done in a single call with the `WasmEdge_VMRunWasmFromFile` convenience function:

```
function WasmEdge_VMRunWasmFromFile(
     Cxt:PWasmEdge_VMContext;
     Path:pcchar;
     FuncName:TWasmEdge_String;
     Params:PWasmEdge_Value;
     ParamLen:Tuint32_t;
     Returns:PWasmEdge_Value;
     ReturnLen:Tuint32_t):TWasmEdge_Result;cdecl;
```

The arguments to this function are pretty straightforward: `path` is the filename of the module to load. `FuncName` is the function to load, `Params` and `ParamLen` specify the parameters that must be passed to the function and `Returns` and `ReturnLen` indicate the location where the return values of the function must be stored.

In the case of the fibonacci function, the function is used as follows:

Figure 4: The 32th fibonacci number calculated by a webassembly program.


```
  pmodule:=pcchar(PAnsiChar(ParamStr(1)));
  Writeln('Running function "fib"')
  Res := WasmEdge_VMRunWasmFromFile(VMCxt, pmodule, FuncName,
                                    @Params, 1,
                                    @Returns, 1);
  if (WasmEdge_ResultOK(Res)) then
    Writeln('Get result: ', WasmEdge_ValueGetI32(Returns[0]))
  else
    Writeln('Error message: ', PAnsiChar(WasmEdge_ResultGetMessage(Res)));
```

After checking the result of the 'run' function with `WasmEdge_ResultOK`, the return value is retrieved from the first element in the `Returns` array. The `WasmEdge_ValueGetI32` is one of the eight functions which can be used to convert a webassembly return value to a pascal native value, in this case a 32-bit integer.

All that is left to do is to clean up the various resources that were allocated:

```
  Writeln('Cleaning up...');
  WasmEdge_VMDelete(VMCxt);
  WasmEdge_ConfigureDelete(ConfCxt);
  WasmEdge_StringDelete(FuncName);
end.
```

The result of this program can be seen in figure 4 on page 16.

6 Embedding a FPC generated program with WasMedge.

To embed a FPC generated webassembly module is not so different from the above program. The start of the program is similar, the differences are in the way the webassembly virtual machine is created.

```
uses ctypes, libwasmedge;

var
  ConfCxt : PWasmEdge_ConfigureContext;
  VMCxt : PWasmEdge_VMContext;
  Returns, Params : Array[0..0] of TWasmEdge_Value;
  FuncName : TWasmEdge_String;
  Res : TWasmEdge_Result;
  pmodule : pcchar;
  WasiModule : PWasmEdge_ModuleInstanceContext;
  ModName : TWasmEdge_String;

begin
  Writeln('Loading library...');
  Loadlibwasmedge('./'+libwasmname);

  Writeln('Adding WASI environment...');
  ConfCxt:=WasmEdge_ConfigureCreate();
  WasmEdge_ConfigureAddHostRegistration(ConfCxt, WasmEdge_HostRegistration_Wasi);
  Writeln('Creating engine...');
  VMCxt:=WasmEdge_VMCreate(ConfCxt,Nil);
```

Until here, there is no difference. Like in the case of the `wasmtime` library, the next step is retrieving an instance of the WASI module and configuring it.

This is done using the `WasmEdge_VMGetImportModuleContext` and `WasmEdge_ModuleInstanceInitWASI` functions. The first of these two functions returns the module context of the predefined WASI module: this predefined module was enabled using the `WasmEdge_ConfigureAddHostRegistration` function, and must be configured with the `WasmEdge_ModuleInstanceInitWASI` procedure:

```
procedure WasmEdge_ModuleInstanceInitWASI(
     Cxt: PWasmEdge_ModuleInstanceContext;
     Args: Ppcchar; ArgLen:Tuint32_t;
     Envs: Ppcchar; EnvLen:Tuint32_t;
     Preopens:Ppcchar; PreopenLen:Tuint32_t);
```

As you can see, the list of command-line parameters, environment variables and allowed directories for file access can be configured. Standard input/output/error cannot be configured as in `wasmtime`. For our case, neither command-line parameters or environment variables are needed, so the configuration is quite simple:

```
WasiModule:=WasmEdge_VMGetImportModuleContext(VMCxt,WasmEdge_HostRegistration_Wasi);
WasmEdge_ModuleInstanceInitWASI(WasiModule,Nil,0,Nil,0,Nil,0);
```

With this we can load our webassembly module and execute the function. We cannot do this with the `WasmEdge_VMRunWasmFromFile` function, instead we need to load the module with the `WasmEdge_VMRegisterModuleFromFile` function:

```
function WasmEdge_VMRegisterModuleFromFile(Cxt: PWasmEdge_VMContext;
                                           ModuleName: TWasmEdge_String;
                                           Path: pcchar): TWasmEdge_Result;
```

The module name must be specified and must be unique. When multiple modules are loaded, then the engine will link together the modules by name. If module 'a'

needs to import function 'b.proc1' then the name 'b' must be provided when loading the webassembly module which contains procedure 'proc1': modules do not have a name associated with them, and the filename is not related to the module name.

Since the module name is passed on as a `TWasmEdge_String` type, we must allocate it with `WasmEdge_StringCreateByCString` before loading the module:

```
ModName:=WasmEdge_StringCreateByCString(Pcchar('prog'));
pmodule:=pcchar(PAnsiChar('hello.wasm'));
Res:=WasmEdge_VMRegisterModuleFromFile(VMCxt, modname, pmodule);
if (WasmEdge_ResultOK(Res)) then
  Writeln('Loaded OK')
else
  Writeln('Error message: ', PAnsiChar(WasmEdge_ResultGetMessage(Res)));
```

Now that the module is loaded, we can actually run the `_start` function:

```
Writeln('Running function "_start"');
FuncName:=WasmEdge_StringCreateByCString(Pcchar('_start'));
Res:=WasmEdge_VMExecuteRegistered(VMCxt, ModName, FuncName,@Params, 0,@Returns,0);
if (WasmEdge_ResultOK(Res)) then
  begin
  Writeln('Run OK')
  Writeln('Exit code: ',WasmEdge_ModuleInstanceWASIGetExitCode(WasiModule));
  end
else
  Writeln('Error message: ', PAnsiChar(WasmEdge_ResultGetMessage(Res)));
```

Note that we retrieved the exit code of the FPC program with the `WasmEdge_ModuleInstanceWASIGetExitCode` function: in difference with `wasmtime`, the `wasmedge` library does not use a trap to set the exit code.

And with that, all that is left to do is clean up, similar to the previous sample program:

```
  Writeln('Cleaning up...');
  WasmEdge_VMDelete(VMCxt);
  WasmEdge_ConfigureDelete(ConfCxt);
  WasmEdge_StringDelete(FuncName);
end.
```

With this, the program can be tested, and the output should look like in figure 5 on page 19

7 Conclusion

In previous articles we've shown that Free Pascal can be used to generate webassembly modules. And as shown in this article, using some external libraries, native FPC programs can load webassembly modules, whether they are generated by FPC or by some other tool.

Figure 5: Using wasmedge to run a FPC-generated webassembly module in a native FPC program.