Getting started with FPC and WebAssembly

Michaël Van Canneyt

December 29, 2021

Abstract

The Free Pascal and Lazarus foundation sponsored development of a WebAssembly backend for FPC. The backend is now usable in production, and we'll show how to work with it in this article

1 Introduction

WebAssembly (Wasm) is gaining traction: Starting out as a way to make Javascript run faster in the browser (Asm.js), it has now become a full description of a runtime engine, designed to run bytecode in a safe way, regardless of where the code is running:

`https://webassembly.org/`

All Major browsers support the running of WebAssembly byte code, Node.JS and Deno.

Not only that, but major languages (C/C++,Rust, C#) - can be compiled to WebAssembly using a special libc library, thus allowing a C#, C/C++ program to run in the browser.

The developers at Mozilla took it even a step further: because WebAssembly is designed to be safe, sensitive parts of the browser are converted to WebAssembly, and then converted back to C++, thus guaranteeing that the resulting code is completely sandboxed and will not be able to penetrate into the rest of the browser.

A webassembly program can now be run in the browser, but also on a server, as part of Javascript runtimes such as Node.JS or Deno, or using a dedicated runtime: wasmtime

`https://wasmtime.dev/`

or wasmer:

`https://wasmer.io/`

Both provide a command-line runtime engine that can load a WebAssembly file and run the code in it. They allow access to the filesystem and interaction with the console through a common API to allow the WebAssembly code to interact with the host environment. This API is called WASI (which is an acronym for WebAssembly System Interface):

`https://wasi.dev/`

Since some time, the Free Pascal compiler can emit Webassembly code, which also relies on the WASI API to talk to the host environment. The WebAssembly backend is meanwhile sufficiently mature to compile many of the packages and units supplied with Free Pascal. The `Goto` statement is not yet implemented, but this is a matter of time before it is implemented.

In this article, we explore how to make use of this new compiler backend.

2 Installation

The Free Pascal WebAssembly compiler is not yet officially released. This means that you must build it yourself if you wish to use it. The Free Pascal WebAssembly compiler makes use of the linker of the LLVM project. So, the first step is to install the LLVM linker. The LLVM linker is part of LLVM, and can be downloaded here for windows:

```
https://github.com/llvm/llvm-project/releases/download/llvmorg-12.0.1/LLVM-12.0.1
```

The installer will ask you if it must add the folder with binaries to the path: you must instruct it to do so.

When it is done, you must copy the application `wasm-ld.exe` to `wasm32-wasi-wasm-ld.exe`, as the latter is what the compiler expects to find.

For linux and MacOS, the package manager can be used to install llvm. For example, on Linux Ubuntu 20.04 this is done using:

```
apt install lld-12
ln -sf /usr/lib/llvm-12/bin/wasm-ld ~/bin/wasm32-wasi-wasm-ld
```

For MacOS, the macports system can be used to install `llvm-12`.

Obviously, you need to have the latest Free Pascal compiler installed. If you have the latest version of the Lazarus IDE installed, then you will have an up-to date compiler installed as well.

The following commands assume that the Free Pascal compiler is installed on your system, and that the `fpc.exe` binary is in your PATH.

Using the installed compiler the Free Pascal webassembly cross-compiler must be built.

This must be done with the latest sources of FPC. somewhere on your system, use git to clone the latest sources (the following must be executed in a command-line window):

```
git clone https://gitlab.com/freepascal.org/fpc/source.git fpc
```

When git has completed the clone operation, build the cross compiler. This can be done with the following commands:

```
cd fpc
make all OS_TARGET=wasi CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-" PP=fpc
cd compiler\utils
make all
cd ..\..
```

If all goes well, you will have built a `ppcrosswasm32.exe` compiler.

This new compiler can be installed with the following command:

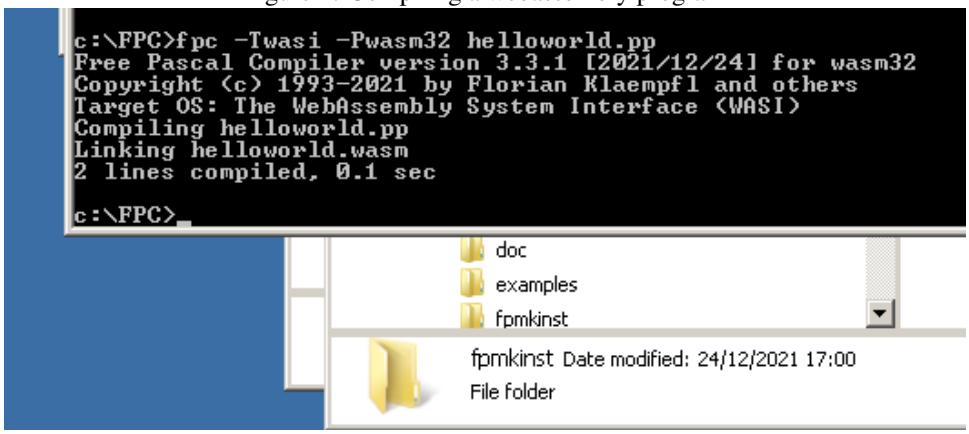
```
make install OS_TARGET=wasi CPU_TARGET=wasm32 BINUTILSPREFIX= OPT="-O-" PP=fpc
cd compiler\utils
make install
cd ..\..
```

This will install a newer version of the `fpc` binary.

More detailed information on building and installing the Free Pascal compiler can be found on

```
https://wiki.freepascal.org/Installing_the_Free_Pascal_Compiler
```

Figure 1: Compiling a webassembly program



3 Compiling for webassembly

Compiling with the Free Pascal webassembly compiler is no different from compiling for any other supported platform. We'll start with the simplest Free Pascal program, which we'll save somewhere in a file called `helloworld.pas`:

```
program helloworld;

begin
  Writeln('Hello, world!');
end.
```

To compile this program from the command-line, the following can be done:

```
fpc -Twasi -Pwasm32 helloworld.pas
```

The compiler will compile and if all went well, you'll see some output as in figure 1 on page 3. Alternatively, the following completely equivalent command can be used:

```
ppcrosswasm32 helloworld.pas
```

To compile for WebAssembly in Lazarus, there are several options, depending on which version of Lazarus you are using.

For all options, you must disable the generation of debug information in the `Project Options` dialog under the page `compiler options - debugging`.

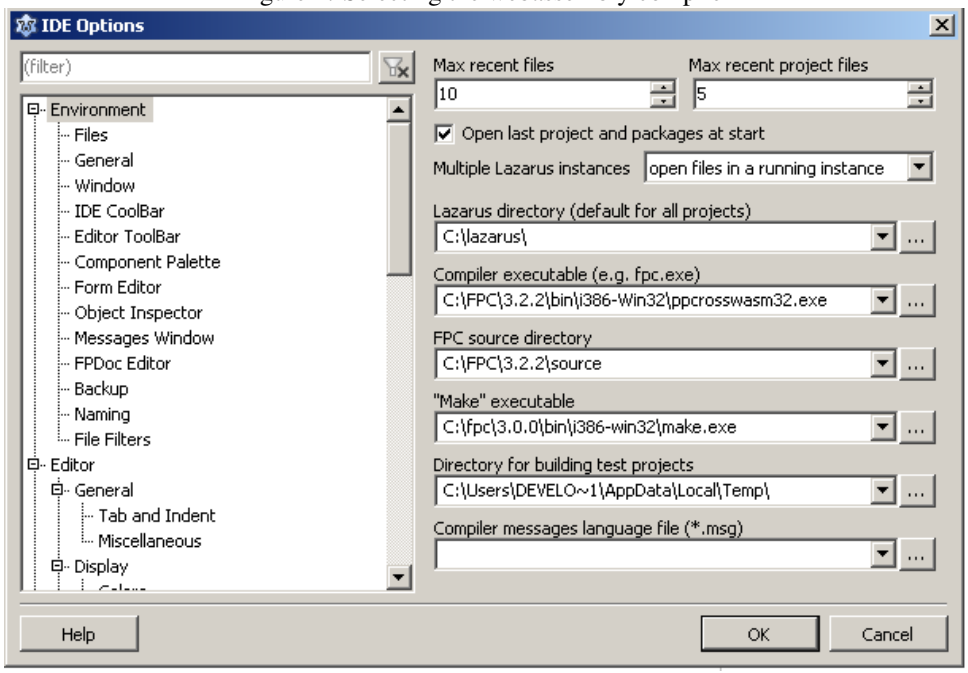
For the officially released version, there are 2 options to choose from. The first one is easiest, but has a drawback: In the `Tools-Options` dialog, select the `ppcrosswasm32.exe` from the following directory:

```
C:\FPC\3.2.2\bin\i386-Win32
```

This is shown in figure 2 on page 4 After doing this, every project you compile will be compiled for WebAssembly. (and that includes the IDE itself if you decide to rebuild it)

Obviously this is normally not desirable, in practice only certain projects will be compilable for WebAssembly. The better way is to use the `Compile commands` from the project options, as shown in figure 3 on page 5.

Figure 2: Selecting the webassembly compiler



The `Execute Before` command can be used to run the cross-compiler. For this all options after `Call on` must be set, and the command must be set to

```
C:\FPC\3.2.2\bin\i386-Win32\ppcrosswasm32 $(ProjFile)
```

You can add any other command-line options that you wish to have. Under `Parsers`, select `FPC`. This tells the IDE to parse the output of the command as it would parse FPC output. Then, under the `Compiler` section, disable all the `Call on` options.

After this, when you compile, it will be as if you compile a program for the native OS on which the IDE is running, see figure 4 on page 5.

If you are using the development version of Lazarus, the above options will still work. However, with the development version it is even easier to compile for webassembly with the development version. It is sufficient to select `wasm32` as the target processor, and `wasi` as the target OS, as shown in figure 5 on page 6 The compiled file will also have the correct extension (`.wasm`) for Lazarus sources of December 28 2021 or later.

4 Running a webassembly program natively

Now that we've succesfully compiled a simple webassembly program, we of course will want to run it. For this, we can use the wasmer or wasmtime command-line WebAssembly runtimes. The runtime command can be downloaded from:

```
https://github.com/bytecodealliance/wasmtime/releases/tag/v0.32.0
```

Once installed, running the generated webassembly is easy. In a command-line terminal, run the following command in the Webassembly project directory:

```
wasmtime helloworld.wasm
```

Figure 3: Using the webassembly compiler for a single project

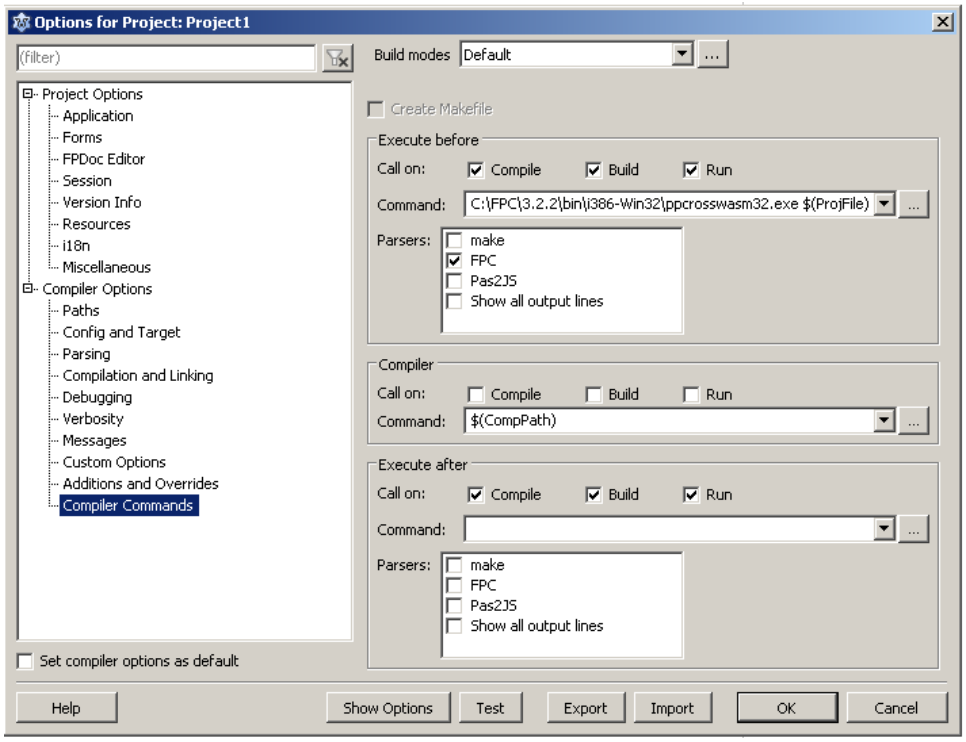


Figure 4: Compilation with the webassembly compiler

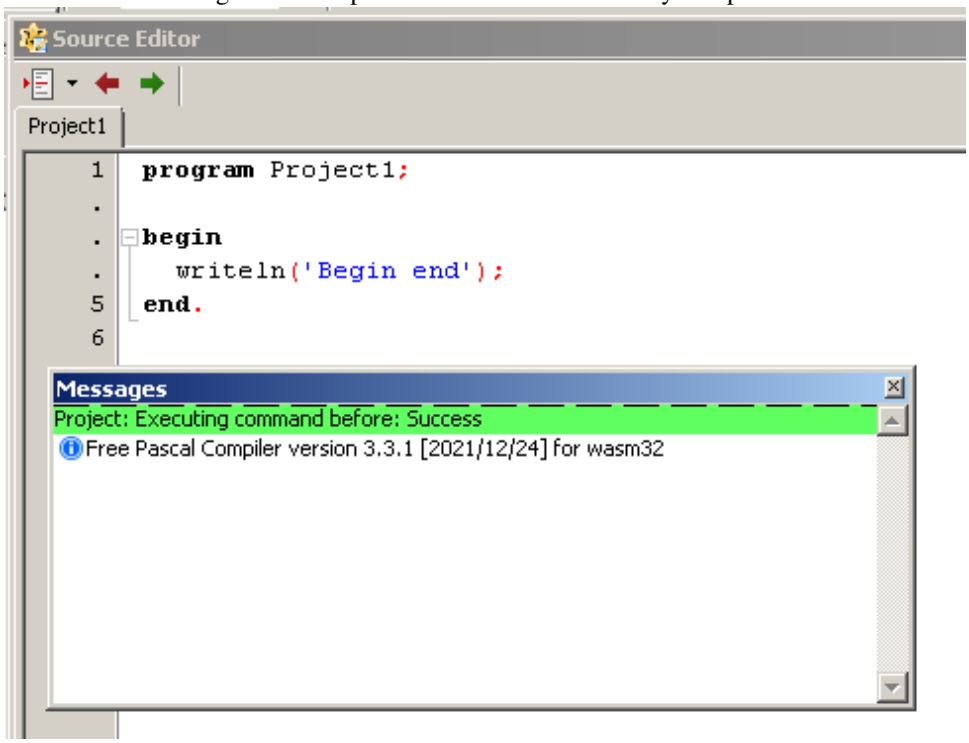


Figure 5: Using the webassembly compiler for a single project

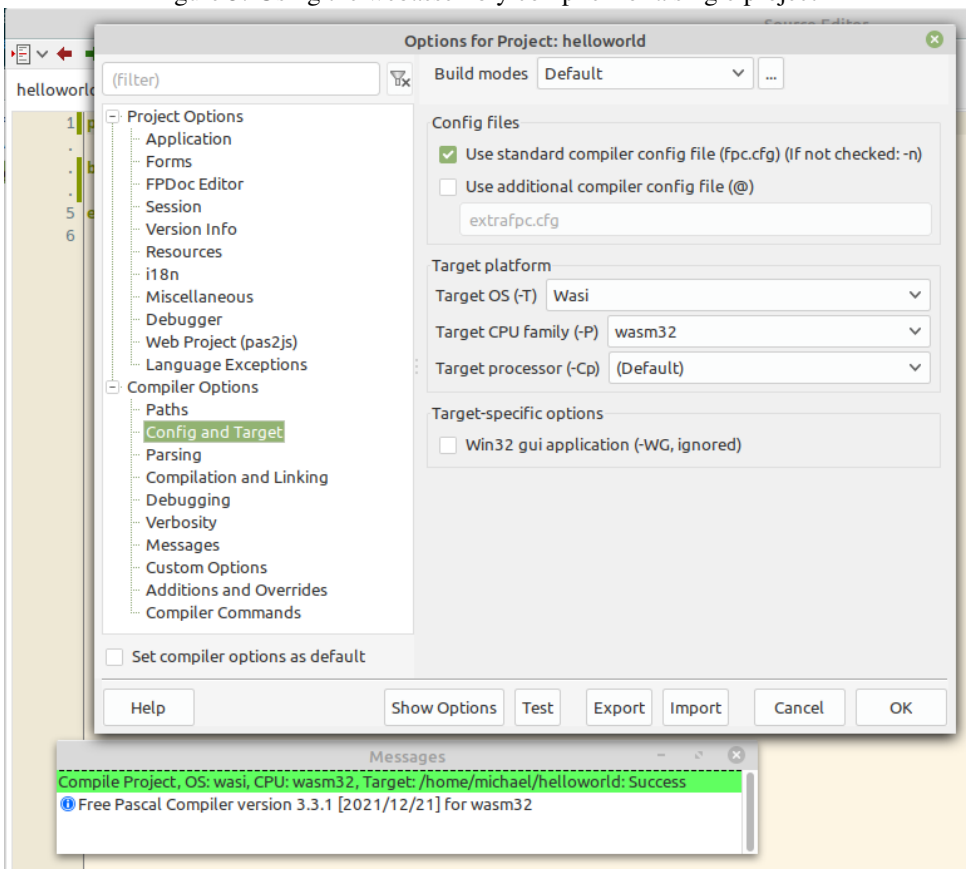
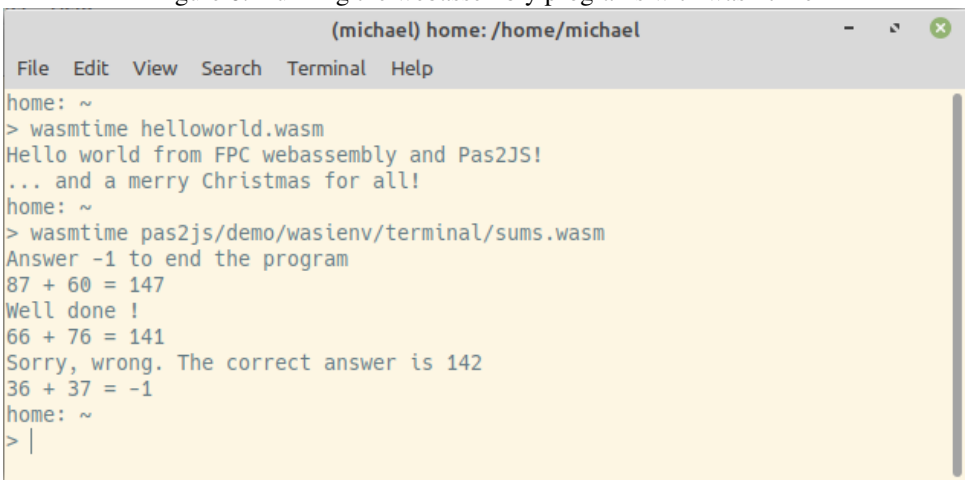


Figure 6: Running the webassembly programs with wasmtime



Or, you can compile and run the `sums.pp` demo project, which is part of the pas2js demos for Webassembly (you can find it in the folder `demos/wasienv/terminal`):

```
ppcrosswasm32 sums.pp
wasmtime sums.wasm
```

And the result will look like figure 6 on page 7, where you can see that the `sums` program actually reads input from the terminal.

5 Running a webassembly program in the browser

A webassembly program can be loaded and run in the browser. The browser offers APIs to do so, and using Pas2JS, you can easily create a hosting environment for your webassembly program. There are currently 2 options to do so:

1. Manually load and run the webassemly file using the provided WASI environment class `TPas2JSWASIEnvironment`.
2. Use the Pas2js-provided `TWASIHostApplication` application class and let it do the heavy lifting for you – it uses the `TPas2JSWASIEnvironment` class in the background.

We'll start with the former method.

Let's start by explaining what the `TPas2JSWASIEnvironment` class is for: WebAssembly standards do not make any assumptions about the environment in which the WebAssembly code is executed. Yet, WebAssembly would not be interesting if it could not interact with the environment. To interact with the outside world, WebAssembly code relies on imported routines: the specifications do specify the mechanism to call external routines.

The FPC RTL for WebAssembly currently follows the WASI standard to interact with the host environment. The WASI standard describes a minimal set of import routines, and is used by the WasmTime and Wasmer runtime environments.

The `TPas2JSWASIEnvironment` class is implemented in Pas2JS, and offers all the callbacks needed for the WebAssembly runtime generated by FPC: These are the callbacks specified by the WASI standard. Although all callbacks are present, they are currently not

all implemented. The class offers also the possibility to hook additional APIs and catch input and output. The following is the public API of this class:

```
TPas2JSWASIEnvironment = class(TObject)
  Function GetUTF8StringFromMem(aLoc, aLen : Longint) : String;
  Procedure AddImports(aObject: TJSObject);
  Property ImportObject : TJSObject;
  Property IsLittleEndian : Boolean;
  Property OnStdOutputWrite : TWASIWriteEvent;
  Property OnStdErrorWrite : TWASIWriteEvent;
  Property OnGetConsoleInputBuffer : TGetConsoleInputBufferEvent;
  Property OnGetConsoleInputString : TGetConsoleInputStringEvent;
  Property Instance : TJSWebAssemblyInstance;
  Property Exitcode : Nativeint;
  // Default is set to the one expected by FPC runtime:
  // wasi_snapshot_preview1
  Property WASIImportName : String;
end;
```

The `GetUTF8StringFromMem` method is a utility call that will retrieve an UTF8 string from the webassembly memory (indicated by a location and length), and returns it as a Javascript string. The `AddImports` call will add the WASI imports to the passed object, as well as any additional APIs you have defined (more about that later). The following properties are also available:

IsLittleEndian A property describing whether the webassembly memory is little-endian or big-endian.

OnStdOutputWrite Called when the WebAssembly program writes to standard output.

OnStdErrorWrite Called when the WebAssembly program writes to standard error.

OnGetConsoleInputBuffer Called when the WebAssembly program tries to read from standard input. Use this event if you wish to pass binary data.

OnGetConsoleInputString Called when the WebAssembly program tries to read from standard input. Use this event if you wish to pass textual data.

Instance This is the currently running `TJSWebAssemblyInstance`.

ExitCode This is the exit code of the WebAssembly program.

WASIImportName This is the name for the import object for the WASI API: the default is `wasi_snapshot_preview1`.

So, how to use this class to run a webassembly file ?

To demonstrate this, we create a small Pas2JS program in the Lazarus IDE (see the article on writing real-world Pas2JS applications on how to get started with Pas2JS), and we instruct the IDE to use the TBrowserApplication for the program source.

In the application class' constructor, we create the WASI environment for our webassembly program:

```
constructor TMyApplication.Create(aOwner: TComponent);
begin
  inherited Create(aOwner);
  FWasiEnv:=TPas2JSWASIEnvironment.Create;
```

```
  FWasiEnv.OnStdErrorWrite:=@DoWrite;
  FWasiEnv.OnStdOutputWrite:=@DoWrite;
end;

procedure TMyApplication.DoWrite(Sender: TObject; const aOutput: String);
begin
  Writeln(aOutput);
end;
```

As you can see, we use the Pascal `Writeln` function to write the standard & error output of the webassembly program. Because we're using the `BrowserConsole` unit, the output will be written in the HTML page.

What was created in the constructor must be destroyed in the destructor, so we implement that too:

```
destructor TMyApplication.Destroy;
begin
  FreeAndNil(FWasiEnv);
  inherited Destroy;
end;
```

The `DoRun` method of the application object must be overriden to implement the actual program logic. In our case, we simply call `InitWebAssembly`:

```
procedure TMyApplication.doRun;

begin
  Terminate;
  InitWebAssembly;
end;
```

The `InitWebAssembly` method is where we set up the WebAssembly environment. The environment for a WebAssembly program is simply a Javascript object that contains various configuration objects as well as routines to be imported in the WebAssembly runtime. You can provide more routines than the environment needs, but all routines that the environment needs must be present in the import object.

Two important (but optional) objects in this regard are

- The `TJSWebAssemblyMemory` object with memory that can be made available to the webassembly runtime.
- a `TJSWebAssemblyTable` object may be specified that will contain a list of callable functions (or imported functions): These are functions that are defined in the WebAssembly module, and which can be called directly from Javascript.

The memory object takes a descriptor record for the constructor. This descriptor specifies the initial and maximum memory for the WebAssembly memory object. The values are specified in WebAssembly pages with 64Kb size. Similarly, the table uses a descriptor which allows to set initial and maximum sizes for the table, and what table you want: the 'anyfunc' value tells the WebAssembly engine to fill the table with all available functions.

```
procedure TMyApplication.InitWebAssembly;

Var
```

```
  mDesc : TJSWebAssemblyMemoryDescriptor;
  tDesc: TJSWebAssemblyTableDescriptor;
  ImportObj : TJSObject;

begin
  //  Setup memory
  mDesc.initial:=256;
  mDesc.maximum:=256;
  FMemory:=TJSWebAssemblyMemory.New(mDesc);
  // Setup table
  tDesc.initial:=0;
  tDesc.maximum:=0;
  tDesc.element:='anyfunc';
  FTable:=TJSWebAssemblyTable.New(tDesc);
  // Setup ImportObject
  ImportObj:=new([
    'js', new([
      'mem', FMemory,
      'tbl', FTable
    ])
  ]);
  FWasiEnv.AddImports(ImportObj);
  CreateWebAssembly('helloworld.wasm',ImportObj)._then(@initEnv)
end;
```

Note that the current implementation of FPC WebAssembly does not import the `js.mem` or `js.tbl` memory objects, but you can import them manually, so the above is just for demonstration purposes in case you wish to use additional memory.

The important call here is to the `FWasiEnv.AddImports` method: this method will add all necessary WASI and additional optional exports to the WebAssembly import object.

After the call to `AddImports`, the `ImportObject` object is ready to be used in the `CreateWebassembly` call: This call returns a promise, which will result in a `TJSInstantiateResult` object. We let the promise resolve in the `InitEnv` method:

```
function TMyApplication.InitEnv(aValue: JSValue): JSValue;

Var
  Module : TJSInstantiateResult absolute aValue;
  exps : TWASIExports;

begin
  Result:=True;
  Exps := TWASIExports(TJSObject(Module.Instance.exports_));
  FWasiEnv.Instance:=Module.Instance;
  Exps.Start;
end;
```

The `Module` variable is just a declaration to avoid typecasts. The `TWASIExports` class is an extension of the `TJSModulesExports` class: this class exports the memory of the WebAssembly object, and contains the `Start` symbol. The `Start` symbol is the name of the program entry point, the only symbol the FPC runtime exports by default.

With this definition, it will be clear that the `Exps.Start` statement actually calls the program's main pascal function (the begin of the program). It's important to realize that this

function does not return as long as the WebAssembly program is running, thus potentially blocking the browser.

The `CreateWebAssembly` call loads the wasm file, and calls all the necessary WebAssembly functions to compile and instantiate the WebAssembly instance:

```
function TMyApplication.CreateWebAssembly(Path: string;
                        ImportObject: TJSObject): TJSPromise;

begin
  Result:=window.fetch(Path)._then(
    Function (res : jsValue) : JSValue
      begin
        Result:=TJSResponse(Res).arrayBuffer._then(
          Function (res2 : jsValue) : JSValue
           begin
             Result:=TJSWebAssembly.instantiate(TJSArrayBuffer(res2),
                                                ImportObject);
           end,
          Nil)
      end,
      Nil);
 end;
```

The following happens:

1. The `Fetch` call will fetch the webassembly file, and retuns a promise.
2. The promise resolves to a `TJSResponse` result, and this is converted to an `TJSArrayBuffer` – this conversion is again returning a promise.
3. The converted array buffer contains the webassembly bytecode wich is then passed to the `TJSWebAssembly.instantiate` function which creates a WebAssembly runtime instance.

The result of the `CreateWebAssembly` function is a promise, which resolves to the result of the `TJSWebAssembly.instantiate` function (a promise in itself). When the instantiated WebAssembly runtime instance is ready, the function resolves to a `TJSInstantiateResult` instance.

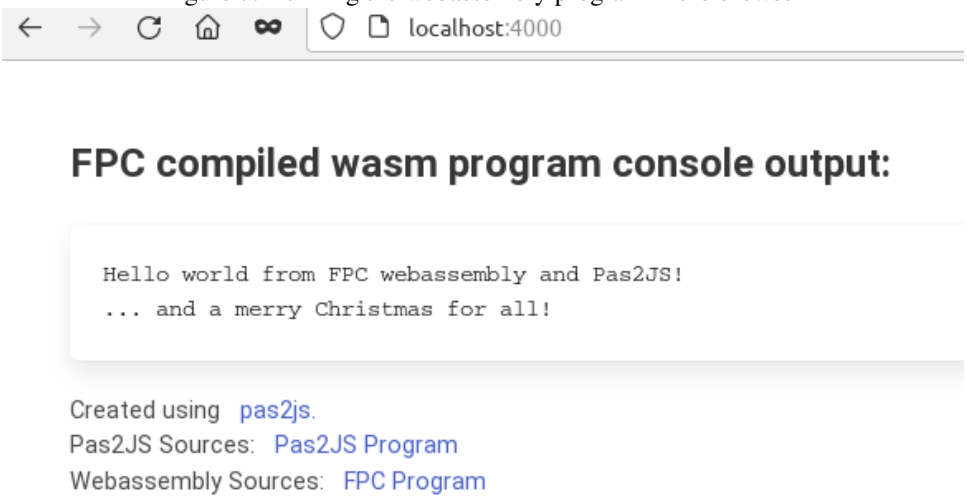
A shorter (and faster) version of this call is:

```
function TMyApplication.CreateWebAssembly(Path: string;
                         ImportObject: TJSObject): TJSPromise;
begin
  Result:=TJSWebAssembly.instantiateStreaming(Fetch(Path),
                                              ImportObject);
end;
```

The difference between `InstantiateStreaming` and `Instantiate` calls is that the former starts compiling the WebAssembly as the bytes come in from the fetch operation. However, you may wish to instantiate a WebAssembly runtime multiple times: in that case it may be better to keep the WebAssembly in memory as soon as it is loaded.

We can add some HTML to beautify the page in which this program is embedded, and after the pas2js program has finished running, this leads to a page such as can be seen figure 7 on page 12. Since the program is loaded as soon as the HTML page is loaded, this is also

Figure 7: Running the webassembly program in the browser



the initial view of the page. You can easily check this by adding a button to the page, and use its `OnClick` event to call the `InitWebAssembly` function.

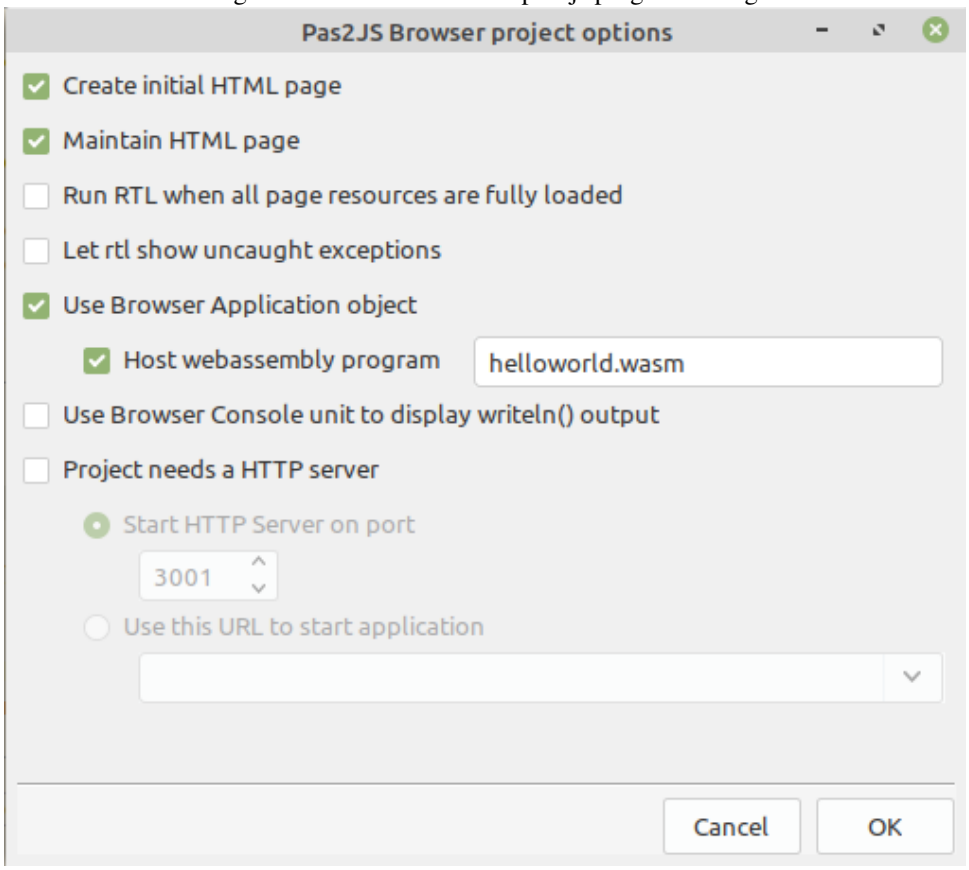
We mentioned earlier that there are 2 ways to run your WebAssembly program. As all long-time users know, Pas2JS and the Lazarus IDE continuously try to make things easier for the developer. That is why `Pas2JS` provides the `TWASIHostApplication` object: a descendent of `TBrowserApplication`.

In the development version of Lazarus, the `New Project` dialog's `Web Browser Application` entry has an additional option called `Host WebAssembly program` (see figure 8 on page 13): When checked, you can enter the URL of the wasm file you wish to load.

In that case, the new project wizard generates the following code, which makes use of the `TWASIHostApplication` class:

```
program Project1;

{$mode objfpc}

uses
 browserapp, wasihostapp, JS, Classes, SysUtils, Web;

type
  TMyApplication = class(TWASIHostApplication)
    procedure doRun; override;
  end;

procedure TMyApplication.doRun;

begin
  StartWebAssembly('helloworld.wasm');
  Terminate;
end;

var
  Application : TMyApplication;
```

Figure 8: The enhanced new pas2js program dialog



```
begin
  Application:=TMyApplication.Create(nil);
  Application.Initialize;
  Application.Run;
end.
```

This project is ready to run: on startup it will use the `StartWebAssembly` method to load the `helloworld.wasm` webassembly in a standard `TPas2JSWASIEnvironment` environment. This environment writes output to the browser console, and input is obtained using the `Prompt` call of the window (a blocking call). The environment can be modified using properties of the application object.

6 Extending the WebAssembly environment

As mentioned before, the WebAssembly specification does not contain an API for interacting with the outside world. In the context of the browser, this means that there is no standard API for accessing the DOM and changing the web page. However, the specification does describe how to import functions. This mechanism is used in the WASI specification to enable low-level access to the host environment: The API caters mainly for file and directory access.

So, if we want to manipulate the webpage in the browser, we'll have to provide an API to the WebAssembly environment. The API can be anything we want: we have complete control over what we allow the WebAssembly environment to do. The `TJSWasiEnvironment` class has support for easily adding additional APIs to the webassembly environment.

This support comes in the form of the `TImportExtension` class. This class serves as a parent class for classes to extend the standard WASI environment. It has the following public declaration:

```
TImportExtension = class (TObject)
Public
  Constructor Create(aEnv : TPas2JSWASIEnvironment); virtual;
  Procedure FillImportObject(aObject : TJSObject); virtual; abstract;
  Function ImportName : String; virtual; abstract;
  Property Env : TPas2JSWASIEnvironment Read FEnv;
end;
```

The constructor has a single argument: the `TPas2JSWASIEnvironment` instance which must be extended; The `TImportExtension` class will register itself in the environment. When the `AddImports` method of the `TPas2JSWASIEnvironment` instance is called to initialize the imports for the webassembly instance, all registered `TImportExtension` classes will be asked to create an import object, which will be added to the import object passed to the WebAssembly instance. The environment is available later in the `Env` property, this will allow the implementation to access the instance.

There are 2 abstract functions which must be implemented by a descendent:

FillImportObject this method must add all import methods to the `aObject` parameter; Note that this object is not the WASI enviroment which is extended: each extension object will be imported with a unique object.

ImportName this is the name that is used to add the object passed in `FillImportObject` to the global WebAssembly impory object.

Now we know how to pass additional functions to a WebAssembly runtime. But how must the code running in the WebAssembly engine import such a function? Well, this happens in exactly the same manner as one would import a function from an external library.

Let's analyse the following call, which is part of the FPC RTL, and is used in the system unit:

```
function __wasi_clock_res_get(
  id: __wasi_clockid_t;
  resolution: P__wasi_timestamp_t
): __wasi_errno_t; external 'wasi_snapshot_preview1' name 'clock_res_get';
```

This declares a function `__wasi_environ_get` which accepts 2 arguments, an ID and a pointer. The key elements here are the `external` and `name` modifiers:

- The `external` specifies the name of the import object in which to find the function (in this case `wasi_snapshot_preview1`).
- The `name` is the name of the function that must be present in the object.

In the `TPas2JSWASIEnvironment` class we find a function called `clock_res_get`:

```
function clock_res_get(clockId, resolution: NativeInt): NativeInt; virtual;
```

Note that the ID (an integer) and resolution (a pointer) are both converted to an integer: the reason is that every address is just an index in the global memory array of the WebAssembly engine.

Seeing that the name of the method is the correct name expected by the WebAssembly runtime, does this mean we can simply attach the `TPas2JSWASIEnvironment` instance to the import object? Unfortunately not.

When the webassembly code calls this function, the `this` variable (known in pascal as `Self`) is empty. That is a problem because all methods (static methods excepted) of a class expect a `Self` pointer. So we must register a function that does supply the `this`.

Fortunately, this is easy. Like all extensions, the functions that `TPas2JSWASIEnvironment` exposes are attached to an object, this happens in the `GetImports` call:

```
procedure TPas2JSWASIEnvironment.GetImports(aImports: TJSObject);

begin
  aImports['args_get']:=@args_get;
  aImports['args_sizes_get']:=@args_sizes_get;
  aImports['clock_res_get']:=@clock_res_get;
  // ...
end;
```

As you can see, the `clock_res_get` is attached to the `aImports` objects with the correct name. The @ operator will bind this to the actual function in the object, so when `clock_res_get` is called, `Self` will be available. Note that because of this, the function name must not necessarily equal the name used in the WebAssembly runtime: the name can always be corrected in the `GetImports` call.

To demonstrate how this can be used, we'll add the possibility to let the webassembly draw on a HTML canvas.

The first thing to do is to create the import functions. We create a unit for this, we'll call it `WebCanvas`. The following is part of the unit:

```
unit webcanvas;

interface

Type
  TCanvasError = longint;
  TCanvasID = longint;
  PCanvasID = ^TCanvasID;

Const
  ECANVAS_SUCCESS     = 0;
  ECANVAS_NOCANVAS    = 1;
  ECANVAS_UNSPECIFIED = -1;

function __webcanvas_allocate(
  SizeX : Longint;
  SIzeY : Longint;
  aID: PCanvasID
): TCanvasError; external 'web_canvas' name 'allocate';

function __webcanvas_moveto(
  aID : TCanvasID;
  X : Longint;
  Y : Longint
):  TCanvasError; external 'web_canvas' name 'moveto';

function __webcanvas_filltext(
  aID : TCanvasID;
  X : Longint;
  Y : Longint;
  aText : PByte;
  aTextLen : Longint
):  TCanvasError; external 'web_canvas' name 'filltext';

// ...

implementation

end.
```

As you can see, there is no implementation for these methods: the implementation will be imported from the Javascript host environment. From the declarations, you can see that these methods must be part of an import object called `web_canvas`.

We can use this to create a canvas class that can be used in the webassembly runtime:

```
TWebCanvas = class(TObject)
private
  FCanvasID : Longint;
  FHeight: Longint;
  FWidth: Longint;
Protected
  Procedure Check(aError : TCanvasError; const aMsg : String = '');
Public
  Constructor Create(aWidth,aHeight : Longint);
```

```
  Procedure moveto(X : Longint;Y : Longint);
  Procedure FillText(X : Longint;Y : Longint;S : UTF8String);
end;
```

The `Check` method serves to check the return of the imported functions: all functions return - an arbitrary convention - an error code.

The `Check` function converts this to an exception:

```
procedure TWebCanvas.Check(aError: TCanvasError; const aMsg: String);
begin
  if aError<>ECANVAS_SUCCESS then
    if aMsg='' then
      Raise Exception.CreateFmt('Canvas Operation failed %d',[aError])
    else
      Raise Exception.CreateFmt('%s : Error code %d',[aMsg,aError]);
end;
```

We can use this function to construct the other methods in the class:

```
constructor TWebCanvas.Create(aWidth, aHeight: Longint);
begin
  Check(__webcanvas_allocate(aWidth,aHeight,@FCanvasID),
        'Failed to create web canvas');
  FWidth:=aWidth;
  FHeight:=aHeight;
end;

procedure TWebCanvas.moveto(X: Longint; Y: Longint);
begin
  Check(__webcanvas_moveto(FCanvasID,X,Y));
end;

procedure TWebCanvas.FillText(X: Longint; Y: Longint; S: UTF8String);
begin
  Check(__webcanvas_filltext(FCanvasID,X,Y,
                             PByte(PAnsichar(S)),Length(S)));
end;
```

As you can see, this is not such difficult code. Note how the string is passed to the imported `__web_canvas_filltext` function: This resembles the way strings are passed to C APIs: as a pointer to a null-terminated memory buffer, and a string length.

The class is now ready be used, and it can be used just as any class would be used in the browser itself:

```
Var
  aCanvas : TWebCanvas;

begin
  aCanvas:=TWebCanvas.Create(150,150);
  With aCanvas do
    try
      BeginPath;
      MoveTo(25,25);
```

```
      LineTo(125,125);
      FillText(8,8,'Greetings on the WebAssembly Canvas!');
    finally
      free;
    end;
end.
```

The complete code can be found in the demos of Pas2JS.

This concludes the WebAssembly side of things. So how do we go about creating the implementation in Javascript? We create a descendent of `TImportExtension` called `TWACanvas` - we present only the relevant calls here:

```
TWACanvas = class(TImportExtension)
Protected
  function GetCanvas(aID : TCanvasID) : TJSCanvasRenderingContext2D;
  function allocate(SizeX, SizeY : Longint; aID: Longint): TCanvasError;
  function moveto(aID : TCanvasID; X,Y : Longint): TCanvasError;
  function FillText(aID : TCanvasID; X,Y : Longint;
                    aText : Longint; aTextLen : Longint ):  TCanvasError;
Public
  Constructor Create(aEnv : TPas2JSWASIEnvironment); override;
  Procedure FillImportObject(aObject : TJSObject); override;
  Function ImportName : String; override;
  Property CanvasParent : TJSHTMLELement;
end;
```

The `ImportName` and `FillImportObject` methods must be overridden and this looks like this:

```
procedure TWACanvas.FillImportObject(aObject: TJSObject);

begin
  aObject['allocate']:=@allocate;
  aObject['moveto']:=@moveto;
  aObject['filltext']:=@FillText;
end;

function TWACanvas.ImportName: String;

begin
  Result:='web_canvas';
end;
```

You can see that the names used are the same names as used to import the functions. This is very important: if one of the names is missing, the Javascript WebAssembly runtime will raise a `LinkError` exception when instantiating the WebAssembly runtime.

The `Allocate` function is called to create a new canvas.

```
function TWACanvas.allocate(SizeX : Longint;
                            SizeY : Longint;
                            aID: Longint): TCanvasError;

Var
```

```
  C : TJSElement;
  V : TJSDataView;
  SID : String;

begin
  C:=window.document.createElement('CANVAS');
  CanvasParent.AppendChild(C);
  Inc(FCurrentID);
  SID:=IntToStr(FCurrentID);
  FCanvases[SID]:=TJSHTMLCanvasElement(c).getcontext('2d');
  V:=getModuleMemoryDataView;
  v.setUint32(aID, FCurrentID, env.IsLittleEndian);
  Result:=ECANVAS_SUCCESS;
end;
```

It creates a new `CANVAS` html element, and attaches it to the HTML Element specified in the `CanvasParent` property. It is then stored with a unique ID in a map with allocated canvas elements, so later on the canvas can be retrieved using this unique ID. This mechanism is arbitrary, in a real-world application, the canvas element to use would probably be communicated to the WebAssembly runtime.

The interesting thing here is how the ID is communciated to the WebAssembly runtime: the WebAssembly definition of the `__webcanvas_allocate` call uses a pointer to an address where the ID must be stored (i.e. it is a `var` parameter):

```
function __webcanvas_allocate(
  SizeX : Longint;
  SIzeY : Longint;
  aID: PCanvasID
) : TCanvasError;
```

The pointer is converted to an integer (the index in memory). The memory of the webassembly runtime is exposed to the Javascript environment. The `getModuleMemoryDataView` call returns the memory as a `TJSDataView` class (a standard Javascript class): in essence an object that can be used to read and write to an underlying array. This is then also how the ID is communicated to the webassembly runtime, it is written directly to the WebAssembly memory using the `setUint32` method of `TJSDataView`.

The `MoveTo` function is actually quite easy. It gets 3 integers as parameters, and does not need memory access. It starts by mapping the canvas ID to an actual canvas renderingcontext, as it was saved by the `Allocate` function:

```
function TWACanvas.moveto(aID : TCanvasID;
                          X : Longint;Y : Longint): TCanvasError;

Var
  C : TJSCanvasRenderingContext2D;

begin
  Result:=ECANVAS_NOCANVAS;
  C:=GetCanvas(aID);
  if Assigned(C) then
    begin
    C.moveto(X,Y);
    Result:=ECANVAS_SUCCESS;
```

```
    end;
end;
```

Note that if no canvas was found, a `ECANVAS_NOCANVAS` error is returned.

Sometimes we must also read from the webassembly memory. This is the case for example when a string must be communicated, for example as shown in the `FillText` method:

```
function TWACanvas.FillText(aID: TCanvasID;
                            X: Longint; Y: Longint;
                            aText: Longint;
                            aTextLen: Longint): TCanvasError;
Var
  C : TJSCanvasRenderingContext2D;
  S : String;
begin
  Result:=ECANVAS_NOCANVAS;
  C:=GetCanvas(aID);
  if Assigned(C) then
    begin
    S:=Env.GetUTF8StringFromMem(aText,aTextLen);
    C.FillText(S,X,Y);
    Result:=ECANVAS_SUCCESS;
    end;
end;
```

The WebAssembly program uses a UTF8-encoded ansistring to communicate a string. The `TPas2JSWASIEnvironment` class has a convenience function that reads an UTF8 string from the WebAssembly memory, given a location and length: `GetUTF8StringFromMem`. This function is used here to retrieve the string to be written on the canvas.

To use this class and have it imported in the WebAssembly runtime, we just need to create it after we have created the environment:

```
FWasiEnv:=TPas2JSWASIEnvironment.Create;
FWasiEnv.OnStdErrorWrite:=@DoWrite;
FWasiEnv.OnStdOutputWrite:=@DoWrite;
FWACanvas:=TWACanvas.Create(FWasiEnv);
FWACanvas.CanvasParent:=GetHTMLElement('canvases');
```

That's all there is to it. To remove the extension, it is sufficient to destroy it.

The result of the test program can be seen on

`https://www.freepascal.org/~michael/pas2js-demos/wasienv/canvas/`

it will look like figure 9 on page 21.

7 Conclusion

In this article, we've shown how Free Pascal can be used to write WebAssembly programs. We also demonstrated how `Pas2JS` can be used to host the WebAssembly program in a browser, and how to extend the WebAssembly environment with custom functions. The compiler support for webassembly is quite stable, but support for the Browser hosting using Pas2JS is quite new (for example standard file support needs still to be added to it), and will surely need some time to mature: we'll report about the progress in future contributions.

Figure 9: The canvas demo