Webassembly

- index
- Compiling mkbitmap to WebAssembly
- What is WebAssembly and where did it come from?

index

layout: collection override:tags: [] date: 2023-06-29 pathName: webassembly

Compiling mkbitmap to WebAssembly

In What is WebAssembly and where did it come from?, I explained how we ended up with the WebAssembly of today. In this article, I will show you my approach of compiling an existing C program, `mkbitmap`, to WebAssembly. It's more complex than the hello world example, as it includes working with files, communicating between the WebAssembly and JavaScript lands, and drawing to a canvas, but it's still manageable enough to not overwhelm you.

The article is written for web developers who want to learn WebAssembly and shows step-by-step how you might proceed if you wanted to compile something like `mkbitmap` to WebAssembly. As a fair warning, not getting an app or library to compile on the first run is completely normal, which is why some of the steps described below ended up not working, so I needed to backtrack and try again differently. The article doesn't show the magic final compilation command as if it had dropped from the sky, but rather describes my actual progress, some frustrations included.

About `mkbitmap`

The `mkbitmap` C program reads an image and applies one or more of the following operations to it, in this order: inversion, highpass filtering, scaling, and thresholding. Each operation can be individually controlled and turned on or off. The principal use of `mkbitmap` is to convert color or grayscale images into a format suitable as input for other programs, particularly the tracing program `potrace` that forms the basis of SVGcode. As a preprocessing tool, `mkbitmap` is particularly useful for converting scanned line art, such as cartoons or handwritten text, to high-resolution bilevel images.

You use `mkbitmap` by passing it a number of options and one or multiple file names. For all details, see the tool's man page:

```
$ mkbitmap [options] [filename...]
```

Cartoon image in color.

The original image (Source).

Cartoon image converted to grayscale after preprocessing.

First scaled, then thresholded: `mkbitmap -f 2 -s 2 -t 0.48` (Source).

Get the code

The first step is to obtain the source code of `mkbitmap`. You can find it on the project's website. At the time of this writing, potrace-1.16.tar.gz is the latest version.

Compile and install locally

The next step is to compile and install the tool locally to get a feeling for how it behaves. The `INSTALL` file contains the following instructions:

1. `cd` to the directory containing the package's source code and type `./configure` to configure the package for your system.
   Running `configure` might take a while. While running, it prints some messages telling which features it is checking for.
2. Type `make` to compile the package.
3. Optionally, type `make check` to run any self-tests that come with the package, generally using the just-built uninstalled binaries.
4. Type `make install` to install the programs and any data files and documentation. When installing into a prefix owned by root, it is recommended that the package be configured and built as a regular user, and only the `make install` phase executed with root privileges.

By following these steps, you should end up with two executables, `potrace` and `mkbitmap`—the latter is the focus of this article. You can verify it worked correctly by running `mkbitmap --version`. Here is the output of all four steps from my machine, heavily trimmed for brevity:

Step 1, `./configure`:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... ./install-sh -c -d
checking for gawk... no
checking for mawk... no
checking for nawk... no
```

```
checking for awk... awk
checking whether make sets $(MAKE)... yes
[…]
config.status: executing libtool commands
```

Step 2, `make`:

```
$ make
/Applications/Xcode.app/Contents/Developer/usr/bin/make  all-recursive
Making all in src
clang -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT main.o -MD -MP -MF .deps/main.Tpo -c -o main.o
main.c
mv -f .deps/main.Tpo .deps/main.Po
[…]
make[2]: Nothing to be done for `all-am'.
```

Step 3, `make check`:

```
$ make check
Making check in src
make[1]: Nothing to be done for `check'.
Making check in doc
make[1]: Nothing to be done for `check'.
[…]
==================================================================================
Testsuite summary for potrace 1.16
==================================================================================
# TOTAL: 8
# PASS:  8
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
==================================================================================
make[1]: Nothing to be done for `check-am'.
```

Step 4, `sudo make install`:

```
$ sudo make install
Password:
```

```
Making install in src
 .././install-sh -c -d '/usr/local/bin'
  /bin/sh ../libtool   --mode=install /usr/bin/install -c potrace mkbitmap '/usr/local/bin'
[…]
make[2]: Nothing to be done for `install-data-am'.
```

To check if it worked, run `mkbitmap --version`:

```
$ mkbitmap --version
mkbitmap 1.16. Copyright (C) 2001-2019 Peter Selinger.
```

If you get the version details, you have successfully compiled and installed `mkbitmap`. Next, make the equivalent of these steps work with WebAssembly.

If you followed the preceding platform compilation steps, run `make clean` before proceeding with the WebAssembly compilation steps to remove previous compile artifacts.

Compile `mkbitmap` to WebAssembly

Emscripten is a tool for compiling C/C++ programs to WebAssembly. Emscripten's Building Projects documentation states the following:

> Building large projects with Emscripten is very easy. Emscripten provides two simple scripts that configure your makefiles to use `emcc` as a drop-in replacement for `gcc`—in most cases the rest of your project's current build system remains unchanged.

The documentation then goes on (a little edited for brevity):

> Consider the case where you normally build with the following commands:
>
> `./configure`
> `make`
>
> To build with Emscripten, you would instead use the following commands:
>
> `emconfigure ./configure`
> `emmake make`

So essentially `./configure` becomes `emconfigure ./configure` and `make` becomes `emmake make`. The following demonstrates how to do this with `mkbitmap`.

Step 0, `make clean`:

```
$ make clean
Making clean in src
 rm -f potrace mkbitmap
test -z "" || rm -f
rm -rf .libs _libs
[…]
rm -f *.lo
```

Step 1, `emconfigure ./configure`:

```
$ emconfigure ./configure
configure: ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... ./install-sh -c -d
checking for gawk... no
checking for mawk... no
checking for nawk... no
checking for awk... awk
[…]
config.status: executing libtool commands
```

Step 2, `emmake make`:

```
$ emmake make
make: make
/Applications/Xcode.app/Contents/Developer/usr/bin/make  all-recursive
Making all in src
/opt/homebrew/Cellar/emscripten/3.1.36/libexec/emcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT
main.o -MD -MP -MF .deps/main.Tpo -c -o main.o main.c
mv -f .deps/main.Tpo .deps/main.Po
[…]
make[2]: Nothing to be done for `all'.
```

If everything went well, there should now be `.wasm` files somewhere in the directory. You can find them by running `find . -name "*.wasm"`:

```
$ find . -name "*.wasm"
./a.wasm
./src/mkbitmap.wasm
./src/potrace.wasm
```

The two last ones look promising, so `cd` into the `src/` directory. There are now also two new corresponding files, `mkbitmap` and `potrace`. For this article, only `mkbitmap` is relevant. The fact that they don't have the `.js` extension is a little confusing, but they are in fact JavaScript files, verifiable with a quick `head` call:

```
$ cd src/
$ head -n 20 mkbitmap
// include: shell.js
// The Module object: Our interface to the outside world. We import
// and export values on it. There are various ways Module can be used:
// 1. Not defined. We create it here
// 2. A function parameter, function(Module) { ..generated code.. }
// 3. pre-run appended it, var Module = {}; ..generated code..
// 4. External script tag defines var Module.
// We need to check if Module already exists (e.g. case 3 above).
// Substitution will be replaced with actual code on later stage of the build,
// this way Closure Compiler will not mangle it (e.g. case 4. above).
// Note that if you want to run closure, and also to use Module
// after the generated code, you will need to define   var Module = {};
// before the code. Then that object will be used in the code, and you
// can continue to use Module afterwards as well.
var Module = typeof Module != 'undefined' ? Module : {};

// --pre-jses are emitted after the Module integration code, so that they can
// refer to Module (if they choose; they can also define Module)
```

Rename the JavaScript file to `mkbitmap.js` by calling `mv mkbitmap mkbitmap.js` (and `mv potrace potrace.js` respectively if you want). Now it's time for the first test to see if it worked by executing the file with Node.js on the command line by running `node mkbitmap.js --version`:

```
$ node mkbitmap.js --version
mkbitmap 1.16. Copyright (C) 2001-2019 Peter Selinger.
```

You have successfully compiled `mkbitmap` to WebAssembly. Now the next step is to make it work in the browser.

`mkbitmap` with WebAssembly in the browser

Copy the `mkbitmap.js` and the `mkbitmap.wasm` files to a new directory called `mkbitmap` and create an `index.html` HTML boilerplate file that loads the `mkbitmap.js` JavaScript file.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>mkbitmap</title>
  </head>
  <body>
    <script src="mkbitmap.js"></script>
  </body>
</html>
```

Start a local server that serves the `mkbitmap` directory and open it in your browser. You should see a prompt that asks you for input. This is as expected, since, according to the tool's man page, "*[i]f no filename arguments are given, then mkbitmap acts as a filter, reading from standard input*", which for Emscripten by default is a `prompt()`.

Serving from the `file:` protocol doesn't work for `.wasm` files, you really need to start a local server.

The mkbitmap app showing a prompt that asks for input.

Prevent automatic execution

To stop `mkbitmap` from executing immediately and instead make it wait for user input, you need to understand Emscripten's `Module` object. `Module` is a global JavaScript object with attributes that Emscripten-generated code calls at various points in its execution. You can provide an implementation of `Module` to control the execution of code. When an Emscripten application starts up, it looks at the values on the `Module` object and applies them.

In the case of `mkbitmap`, set `Module.noInitialRun` to `true` to prevent the initial run that caused the prompt to appear. Create a script called `script.js`, include it *before* the `<script src="mkbitmap.js"></script>` in `index.html` and add the following code to `script.js`. When you now reload the app, the prompt should be gone.

```
var Module = {
  // Don't run main() at page load
  noInitialRun: true,
};
```

Create a modular build with some more build flags

To provide input to the app, you can use Emscripten's file system support in `Module.FS`. The Including File System Support section of the documentation states:

> Emscripten decides whether to include file system support automatically. Many programs don't need files, and file system support is not negligible in size, so Emscripten avoids including it when it doesn't see a reason to. That means that if your C/C++ code does not access files, then the `FS` object and other file system APIs will not be included in the output. And, on the other hand, if your C/C++ code does use files, then file system support will be automatically included.

Unfortunately `mkbitmap` is one of the cases where Emscripten does not automatically include file system support, so you need to explicitly tell it to do so. This means you need to follow the `emconfigure` and `emmake` steps described previously, with a couple more flags set via a `CFLAGS` argument. The following flags may come in useful for other projects, too.

When looking at the following commands, you might wonder about the missing space after `-s` that you may see in other online tutorials. Some `-s` options may require quoting, or the space between `-s` and the next argument may confuse `CMake`. To avoid those problems, I recommend you generally use the `-sX=Y` notation, that is, without a space.

- Set `-sFILESYSTEM=1` so file system support is included.
- Set `-sEXPORTED_RUNTIME_METHODS=FS,callMain` so `Module.FS` and `Module.callMain` are exported.
- Set `-sMODULARIZE=1` and `-sEXPORT_ES6` to generate a modern ES6 module.
- Set `-sINVOKE_RUN=0` to prevent the initial run that caused the prompt to appear.

Also, in this particular case, you need to set the `--host` flag to `wasm32` to tell the `configure` script that you are compiling for WebAssembly.

The final `emconfigure` command looks like this:

```
$ emconfigure ./configure --host=wasm32 CFLAGS='-sFILESYSTEM=1 -
sEXPORTED_RUNTIME_METHODS=FS,callMain -sMODULARIZE=1 -sEXPORT_ES6 -sINVOKE_RUN=0'
```

Do not forget to run `emmake make` again and copy the freshly created files over to the `mkbitmap` folder.

Modify `index.html` so that it only loads the ES module `script.js`, from which you then import the `mkbitmap.js` module.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>mkbitmap</title>
  </head>
  <body>
    <!-- No longer load `mkbitmap.js` here -->
    <script src="script.js" type="module"></script>
  </body>
</html>
```

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';

const run = async () => {
  const Module = await loadWASM();
  console.log(Module);
};

run();
```

When you open the app now in the browser, you should see the `Module` object logged to the DevTools console, and the prompt is gone, since the `main()` function of `mkbitmap` is no longer called at the start.

The mkbitmap app with a white screen, showing the Module object logged to the DevTools console.

Manually execute the main function

The next step is to manually call `mkbitmap`'s `main()` function by running `Module.callMain()`. The `callMain()` function takes an array of arguments, which match one-by-one what you would pass on

the command line. If on the command line you would run `mkbitmap -v`, you would call `Module.callMain(['-v'])` in the browser. This logs the `mkbitmap` version number to the DevTools console.

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';

const run = async () => {
  const Module = await loadWASM();
  Module.callMain(['-v']);
};

run();
```

The mkbitmap app with a white screen, showing the mkbitmap version number logged to the DevTools console.

Redirect the standard output

The standard output (`stdout`) by default is the console. However, you can redirect it to something else, for example, a function that stores the output to a variable. This means you can add the output to the HTML by setting the `Module.print` property.

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';

const run = async () => {
  let consoleOutput = 'Powered by ';
  const Module = await loadWASM({
    print: (text) => (consoleOutput += text),
  });
  Module.callMain(['-v']);
  document.body.textContent = consoleOutput;
};

run();
```

The mkbitmap app showing the mkbitmap version number.

Get the input file into the memory file system

To get the input file into the memory file system, you need the equivalent of `mkbitmap filename` on the command line. To understand how I approach this, first some background on how `mkbitmap` expects its input and creates its output.

Supported input formats of `mkbitmap` are PNM (PBM, PGM, PPM) and BMP. The output formats are PBM for bitmaps, and PGM for graymaps. If a `filename` argument is given, `mkbitmap` will by default create an output file whose name is obtained from the input file name by changing its suffix to `.pbm`. For example, for the input file name `example.bmp`, the output file name would be `example.pbm` .

Emscripten provides a virtual file system that simulates the local file system, so that native code using synchronous file APIs can be compiled and run with little or no change. For `mkbitmap` to read an input file as if it was passed as a `filename` command line argument, you need to use the `FS` object that Emscripten provides.

The `FS` object is backed by an in-memory file system (commonly referred to as MEMFS) and has a `writeFile()` function that you use to write files to the virtual file system. You use `writeFile()` as shown in the following code sample.

To verify the file write operation worked, run the `FS` object's `readdir()` function with the parameter `'/'`. You will see `example.bmp` and a number of default files that are always created automatically.

Note that the previous call to `Module.callMain(['-v'])` for printing the version number was removed. This is due to the fact that `Module.callMain()` is a function that generally expects to only be run once.

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';

const run = async () => {
  const Module = await loadWASM();
  const buffer = await fetch('https://example.com/example.bmp').then((res) =>
res.arrayBuffer());
  Module.FS.writeFile('example.bmp', new Uint8Array(buffer));
  console.log(Module.FS.readdir('/'));
};
```

```
run();
```

The mkbitmap app showing an array of files in the memory file system, including example.bmp.

First actual execution

With everything in place, execute `mkbitmap` by running `Module.callMain(['example.bmp'])`. Log the contents of the MEMFS' `'/'` folder, and you should see the newly created `example.pbm` output file next to the `example.bmp` input file.

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';

const run = async () => {
  const Module = await loadWASM();
  const buffer = await fetch('https://example.com/example.bmp').then((res) =>
res.arrayBuffer());
  Module.FS.writeFile('example.bmp', new Uint8Array(buffer));
  Module.callMain(['example.bmp']);
  console.log(Module.FS.readdir('/'));
};

run();
```

The mkbitmap app showing an array of files in the memory file system, including example.bmp and example.pbm.

Get the output file out of the memory file system

The `FS` object's `readFile()` function enables getting the `example.pbm` created in the last step out of the memory file system. The function returns a `Uint8Array` that you convert to a `File` object and save to disk, as browsers don't generally support PBM files for direct in-browser viewing. (There are more elegant ways to save a file, but using a dynamically created `<a download>` is the most widely supported one.) Once the file is saved, you can open it in your favorite image viewer.

```
// This is `script.js`.
import loadWASM from './mkbitmap.js';
```

```
const run = async () => {
  const Module = await loadWASM();
  const buffer = await fetch('https://example.com/example.bmp').then((res) =>
res.arrayBuffer());
  Module.FS.writeFile('example.bmp', new Uint8Array(buffer));
  Module.callMain(['example.bmp']);
  const output = Module.FS.readFile('example.pbm', { encoding: 'binary' });
  const file = new File([output], 'example.pbm', {
    type: 'image/x-portable-bitmap',
  });
  const a = document.createElement('a');
  a.href = URL.createObjectURL(file);
  a.download = file.name;
  a.click();
};

run();
```

macOS Finder with a preview of the input .bmp file and the output .pbm file.

Add an interactive UI

To this point, the input file is hardcoded and `mkbitmap` runs with default parameters. The final step is to let the user dynamically select an input file, tweak the `mkbitmap` parameters, and then run the tool with the selected options.

```
// Corresponds to `mkbitmap -o output.pbm input.bmp -s 8 -3 -f 4 -t 0.45`.
Module.callMain(['-o', 'output.pbm', 'input.bmp', '-s', '8', '-3', '-f', '4', '-t', '0.45']);
```

The PBM image format is not particularly hard to parse, so with some JavaScript code, you could even show a preview of the output image. See the source code of the embedded demo below for one way to do this.

https://glitch.com/edit/#!/mkbitmap

Conclusion

Congratulations, you have successfully compiled `mkbitmap` to WebAssembly and made it work in the browser! There were some dead ends and you had to compile the tool more than once until it worked, but as I wrote above, that's part of the experience. Also remember the StackOverflow's `webassembly` tag if you get stuck. Happy compiling!

Acknowledgements

This article was reviewed by Sam Clegg and Rachel Andrew.

What is WebAssembly and where did it come from?

Ever since the web became a platform not just for documents but also for apps, some of the most advanced apps have pushed web browsers to their limits. The approach of going "closer to the metal" by interfacing with lower-level languages in order to improve performance is encountered in many higher-level languages. As an example, Java has the Java Native Interface. For JavaScript, this lower-level language is WebAssembly. In this article, you will discover what assembly language is, and why it can be useful on the web, then learn how WebAssembly was created via the interim solution of asm.js.

Assembly language

Have you ever programmed in assembly language? In computer programming, assembly language, often referred to simply as Assembly and commonly abbreviated as ASM or asm, is *any* low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

For example, looking at the Intel® 64 and IA-32 Architectures (PDF), the `MUL` instruction (for **mul** tiplication) performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand), and stores the result in the destination operand. Very simplified, the destination operand is an implied operand located in register `AX`, and the source operand is located in a general-purpose register like `CX`. The result is stored again in register `AX`. Consider the following x86 code example:

```
mov ax, 5  ; Set the value of register AX to 5.
mov cx, 10 ; Set the value of register CX to 10.
mul cx     ; Multiply the value of register AX (5)
           ; and the value of register CX (10), and
           ; store the result in register AX.
```

For comparison, if tasked with the objective of multiplying 5 and 10, you would probably write code similar to the following in JavaScript:

```
const factor1 = 5;
const factor2 = 10;
```

```
const result = factor1 * factor2;
```

The advantage of going the assembly route is that such low-level and machine-optimized code is much more efficient than high-level and human-optimized code. In the preceding case it doesn't matter, but you can imagine that for more complex operations, the difference can be significant.

As the name suggests, x86 code is dependent on the x86 architecture. What if there were a way of writing assembly code that was not dependent on a specific architecture, but that would inherit the performance benefits of assembly?

asm.js

The first step to writing assembly code with no architecture dependencies was asm.js, a strict subset of JavaScript that could be used as a low-level, efficient target language for compilers. This sub-language effectively described a sandboxed virtual machine for memory-unsafe languages like C or C++. A combination of static and dynamic validation allowed JavaScript engines to employ an ahead-of-time (AOT) optimizing compilation strategy for valid asm.js code. Code written in statically-typed languages with manual memory management (such as C) was translated by a source-to-source compiler such as the early Emscripten (based on LLVM).

Performance was improved by limiting language features to those amenable to AOT. Firefox 22 was the first browser to support asm.js, released under the name OdinMonkey. Chrome added asm.js support in version 61. While asm.js still works in browsers, it has been superseded by WebAssembly. The reason to use asm.js at this point would be as an alternative for browsers that don't have WebAssembly support.

WebAssembly

WebAssembly is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ and Rust, and many more with a compilation target so that they run on the web. Support for memory-managed languages such as Java, Kotlin, and Dart is in the works and should become available soon. WebAssembly is designed to run alongside JavaScript, allowing both to work together.

Apart from the browser, WebAssembly programs also run in other runtimes thanks to WASI, the WebAssembly System Interface, a modular system interface for WebAssembly. WASI is created to be portable across operating systems, with the objective of being secure and the ability to run in a sandboxed environment.

WebAssembly code (binary code, that is, bytecode) is intended to be run on a portable virtual stack machine (VM). The bytecode is designed to be faster to parse and execute than JavaScript and to have a compact code representation.

Conceptual execution of instructions proceeds by way of a traditional program counter that advances through the instructions. In practice, most Wasm engines compile the Wasm bytecode to machine code, and then execute that. Instructions fall into two categories:

- **Control instructions** that form control constructs and pop their argument value(s) off the stack, may change the program counter, and push result value(s) onto the stack.
- **Simple instructions** that pop their argument value(s) from the stack, apply an operator to the values, and then push the result value(s) onto the stack, followed by an implicit advancement of the program counter.

Going back to the example before, the following WebAssembly code would be equivalent to the x86 code from the beginning of the article:

```
i32.const 5  ; Push the integer value 5 onto the stack.
i32.const 10 ; Push the integer value 10 onto the stack.
i32.mul      ; Pop the two most recent items on the stack,
             ; multiply them, and push the result onto the stack.
```

While asm.js is implemented all in software, that is, its code can run in any JavaScript engine (even if unoptimized), WebAssembly required new functionality that all browser vendors agreed on.

Announced in 2015 and first released in March 2017, WebAssembly became a W3C recommendation on December 5, 2019. The W3C maintains the standard with contributions from all major browser vendors and other interested parties. Since 2017, browser support is universal.

WebAssembly has two representations: textual and binary. What you see above is the textual representation.

Textual representation

The textual representation is based on S-expressions and commonly uses the file extension `.wat` (for **W**eb**A**ssembly **t**ext format). If you really wanted to, you could write it by hand. Taking the multiplication example from above and making it more useful by no longer hardcoding the factors, you can probably make sense of the following code:

```
(module
  (func $mul (param $factor1 i32) (param $factor2 i32) (result i32)
    local.get $factor1
    local.get $factor2
```

```
    i32.mul)
  (export "mul" (func $mul))
)
```

Binary representation

The binary format that uses the file extension `.wasm` is not meant for human consumption, let alone human creation. Using a tool like wat2wasm, you can convert the above code to the following binary representation. (The comments are not usually part of the binary representation, but added by the wat2wasm tool for better understandability.)

```
0000000: 0061 736d                                 ; WASM_BINARY_MAGIC
0000004: 0100 0000                                 ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                                        ; section code
0000009: 00                                        ; section size (guess)
000000a: 01                                        ; num types
; func type 0
000000b: 60                                        ; func
000000c: 02                                        ; num params
000000d: 7f                                        ; i32
000000e: 7f                                        ; i32
000000f: 01                                        ; num results
0000010: 7f                                        ; i32
0000009: 07                                        ; FIXUP section size
; section "Function" (3)
0000011: 03                                        ; section code
0000012: 00                                        ; section size (guess)
0000013: 01                                        ; num functions
0000014: 00                                        ; function 0 signature index
0000012: 02                                        ; FIXUP section size
; section "Export" (7)
0000015: 07                                        ; section code
0000016: 00                                        ; section size (guess)
0000017: 01                                        ; num exports
0000018: 03                                        ; string length
0000019: 6d75 6c                             mul   ; export name
000001c: 00                                        ; export kind
000001d: 00                                        ; export func index
```

```
0000016: 07                                        ; FIXUP section size
; section "Code" (10)
000001e: 0a                                        ; section code
000001f: 00                                        ; section size (guess)
0000020: 01                                        ; num functions
; function body 0
0000021: 00                                        ; func body size (guess)
0000022: 00                                        ; local decl count
0000023: 20                                        ; local.get
0000024: 00                                        ; local index
0000025: 20                                        ; local.get
0000026: 01                                        ; local index
0000027: 6c                                        ; i32.mul
0000028: 0b                                        ; end
0000021: 07                                        ; FIXUP func body size
000001f: 09                                        ; FIXUP section size
; section "name"
0000029: 00                                        ; section code
000002a: 00                                        ; section size (guess)
000002b: 04                                        ; string length
000002c: 6e61 6d65                           name  ; custom section name
0000030: 01                                        ; name subsection type
0000031: 00                                        ; subsection size (guess)
0000032: 01                                        ; num names
0000033: 00                                        ; elem index
0000034: 03                                        ; string length
0000035: 6d75 6c                              mul  ; elem name 0
0000031: 06                                        ; FIXUP subsection size
0000038: 02                                        ; local name type
0000039: 00                                        ; subsection size (guess)
000003a: 01                                        ; num functions
000003b: 00                                        ; function index
000003c: 02                                        ; num locals
000003d: 00                                        ; local index
000003e: 07                                        ; string length
000003f: 6661 6374 6f72 31                 factor1  ; local name 0
0000046: 01                                        ; local index
0000047: 07                                        ; string length
0000048: 6661 6374 6f72 32                 factor2  ; local name 1
```

```
0000039: 15                                        ; FIXUP subsection size
000002a: 24                                        ; FIXUP section size
```

Compiling to WebAssembly

As you see, neither `.wat` nor `.wasm` are particularly very human-friendly. This is where a compiler like Emscripten comes into play. It lets you compile from higher-level languages like C and C++. There are other compilers for other languages like Rust and many more. Consider the following C code:

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

Usually, you would compile this C program with the compiler `gcc`.

```
$ gcc hello.c -o hello
```

With Emscripten installed, you compile it to WebAssembly using the `emcc` command and almost the same arguments:

```
$ emcc hello.c -o hello.html
```

This will create a `hello.wasm` file and the HTML wrapper file `hello.html`. When you serve the file `hello.html` from a web server, you will see `"Hello World"` printed to the DevTools console.

There's also a way to compile to WebAssembly without the HTML wrapper:

```
$ emcc hello.c -o hello.js
```

As before, this will create a `hello.wasm` file, but this time a `hello.js` file instead of the HTML wrapper. To test, you run the resulting JavaScript file `hello.js` with, for example, Node.js:

```
$ node hello.js
Hello World
```

Learn more

This brief introduction to WebAssembly is just the tip of the iceberg. Learn more about WebAssembly in the WebAssembly documentation on MDN and consult the Emscripten documentation. Truth be told, working with WebAssembly can feel a bit like the How to draw an owl meme, especially since web developers familiar with HTML, CSS, and JavaScript are not necessarily versed in the to-be-compiled-from languages like C. Luckily there are channels like StackOverflow's `webassembly` tag where experts are often happy to help if you ask nicely.

Move on to the article Compiling `mkbitmap` to WebAssembly for a beginner-friendly introduction to compiling a not completely trivial but also not overly complex C program to WebAssembly. At the example of `mkbitmap`, this article shows how to use a Wasm program as a library in JavaScript that works with files as input and that outputs images.

Acknowledgements

This article was reviewed by Jakob Kummerow, Derek Schuff, and Rachel Andrew.