

Understanding WebAssembly text format

To enable WebAssembly to be read and edited by humans, there is a textual representation of the Wasm binary format. This is an intermediate form designed to be exposed in text editors, browser developer tools, etc. This article explains how that text format works, in terms of the raw syntax, and how it is related to the underlying bytecode it represents — and the wrapper objects representing Wasm in JavaScript.

Note: This is potentially overkill if you are a web developer who just wants to load a Wasm module into a page and use it in your code (see [Using the WebAssembly JavaScript API](#)), but it is more useful if for example, you want to write Wasm modules to optimize the performance of your JavaScript library, or build your own WebAssembly compiler.

S-expressions

In both the binary and textual formats, the fundamental unit of code in WebAssembly is a module. In the text format, a module is represented as one big S-expression. S-expressions are a very old and very simple textual format for representing trees, and thus we can think of a module as a tree of nodes that describe the module's structure and its code. Unlike the Abstract Syntax Tree of a programming language, though, WebAssembly's tree is pretty flat, mostly consisting of lists of instructions.

First, let's see what an S-expression looks like. Each node in the tree goes inside a pair of parentheses — `(...)`. The first label inside the parenthesis tells you what type of node it is, and after that there is a space-separated list of either attributes or child nodes. So that means the WebAssembly S-expression:

```
(module (memory 1) (func))
```

represents a tree with the root node "module" and two child nodes, a "memory" node with the attribute "1" and a "func" node. We'll see shortly what these nodes actually mean.

The simplest module

Let's start with the simplest, shortest possible Wasm module.

WASM



```
(module)
```

This module is totally empty, but is still a valid module.

If we convert our module to binary now (see [Converting WebAssembly text format to Wasm](#)), we'll see just the 8 byte module header described in the [binary format](#)

:

WASM



```
00000000: 0061 736d          ; WASM_BINARY_MAGIC
00000004: 0100 0000          ; WASM_BINARY_VERSION
```

Adding functionality to your module

Ok, that's not very interesting, let's add some executable code to this module.

All code in a webassembly module is grouped into functions, which have the following pseudocode structure:

WASM



```
( func <signature> <locals> <body> )
```


- The **signature** declares what the function takes (parameters) and returns (return values).
- The **locals** are like vars in JavaScript, but with explicit types declared.

- The **body** is just a linear list of low-level instructions.

So this is similar to functions in other languages, even if it looks different because it is an S-expression.

Signatures and parameters


The signature is a sequence of parameter type declarations followed by a list of return type declarations. It is worth noting here that:

- The absence of a `(result)` means the function doesn't return anything.
- In the current iteration, there can be at most 1 return type, but [later this will be relaxed](#)  to any number.

Each parameter has a type explicitly declared; Wasm [Number types](#), [Reference types](#), [Vector types](#). The number types are:

- `i32` : 32-bit integer
- `i64` : 64-bit integer
- `f32` : 32-bit float
- `f64` : 64-bit float

A single parameter is written `(param i32)` and the return type is written `(result i32)`, hence a binary function that takes two 32-bit integers and returns a 64-bit float would be written like this:

WASM	
<code>(func (param i32) (param i32) (result f64) ...)</code>	

After the signature, locals are listed with their type, for example `(local i32)`. Parameters are basically just locals that are initialized with the value of the corresponding argument passed by the caller.

Getting and setting locals and parameters

Locals/parameters can be read and written by the body of the function with the `local.get` and `local.set` instructions.


The `local.get` / `local.set` commands refer to the item to be got/set by its numeric index: parameters are referred to first, in order of their declaration, followed by locals in order of their declaration. So given the following function:

```
WASM   
(func (param i32) (param f32) (local f64)  
  local.get 0  
  local.get 1  
  local.get 2)
```

The instruction `local.get 0` would get the i32 parameter, `local.get 1` would get the f32 parameter, and `local.get 2` would get the f64 local.

There is another issue here — using numeric indices to refer to items can be confusing and annoying, so the text format allows you to name parameters, locals, and most other items by including a name prefixed by a dollar symbol (`$`) just before the type declaration.

Thus, you could rewrite our previous signature like so:

```
WASM   
(func (param $p1 i32) (param $p2 f32) (local $loc f64) ...)
```


And then could write `local.get $p1` instead of `local.get 0`, etc. (Note that when this text gets converted to binary, though, the binary will contain only the integer.)

Stack machines

Before we can write a function body, we have to talk about one more thing: **stack machines**. Although the browser compiles it to something more efficient, Wasm execution is defined in terms of a stack machine where the basic idea is that every type of instruction pushes and/or pops a certain number of `i32` / `i64` / `f32` / `f64` values to/from a stack.

For example, `local.get` is defined to push the value of the local it read onto the stack, and `i32.add` pops two `i32` values (it implicitly grabs the previous two values pushed onto the stack), computes their sum (modulo 2^{32}) and pushes the resulting `i32` value.

When a function is called, it starts with an empty stack which is gradually filled up and emptied as the body's instructions are executed. So for example, after executing the following function:


```
WASM   
(func (param $p i32)  
  (result i32)  
  local.get $p  
  local.get $p  
  i32.add)
```

The stack contains exactly one `i32` value — the result of the expression `($p + $p)`, which is handled by `i32.add`. The return value of a function is just the final value left on the stack.


The WebAssembly validation rules ensure the stack matches exactly: if you declare a `(result f32)`, then the stack must contain exactly one `f32` at the end. If there is no result type, the stack must be empty.

Our first function body

As mentioned before, the function body is a list of instructions that are followed as the function is called. Putting this together with what we have already learned, we can finally define a module containing our own simple function:

```
WASM   
(module  
  (func (param $lhs i32) (param $rhs i32) (result i32)  
    local.get $lhs  
    local.get $rhs  
    i32.add))
```

This function gets two parameters, adds them together, and returns the result.

There are a lot more things that can be put inside function bodies, but we will start off simple for now, and you'll see a lot more examples as you go along. For a full list of the available opcodes, consult the [webassembly.org Semantics reference](https://webassembly.org/Semantics/reference) .


Calling the function

Our function won't do very much on its own — now we need to call it. How do we do that? Like in an ES module, Wasm functions must be explicitly exported by an `export` statement inside the module.

Like locals, functions are identified by an index by default, but for convenience, they can be named. Let's start by doing this — first, we'll add a name preceded by a dollar sign, just after the `func` keyword:


```
WASM   
(func $add ...)
```

Now we need to add an export declaration — this looks like so:

```
WASM   
(export "add" (func $add))
```

Here, `add` is the name the function will be identified by in JavaScript whereas `$add` picks out which WebAssembly function inside the Module is being exported.

So our final module (for now) looks like this:

```
WASM   
(module  
  (func $add (param $lhs i32) (param $rhs i32) (result i32)  
    local.get $lhs  
    local.get $rhs  
    i32.add)  
  (export "add" (func $add))  
)
```

If you want to follow along with the example, save the above our module into a file called `add.wat`, then convert it into a binary file called `add.wasm` using `wabt` (see [Converting WebAssembly text format to Wasm](#) for details).

Next, we'll asynchronously instantiate our binary (see [Loading and running WebAssembly code](#)) and execute our `add` function in JavaScript (we can now find `add()` in the [exports](#) property of the instance):

```
JS
WebAssembly.instantiateStreaming(fetch("add.wasm")).then((obj) => {
  console.log(obj.instance.exports.add(1, 2)); // "3"
});
```

Note: You can find this example in GitHub as [add.html](#) (see it live also). Also see [WebAssembly.instantiateStreaming\(\)](#) for more details about the instantiate function.

Exploring fundamentals

Now we've covered the real basics, let's move on to look at some more advanced features.

Calling functions from other functions in the same module

The `call` instruction calls a single function, given its index or name. For example, the following module contains two functions — one just returns the value 42, the other returns the result of calling the first plus one:

```
WASM
(module
  (func $getAnswer (result i32)
    i32.const 42)
  (func (export "getAnswerPlus1") (result i32)
    call $getAnswer
    i32.const 1
    i32.add))
```

Note: `i32.const` just defines a 32-bit integer and pushes it onto the stack. You could swap out the `i32` for any of the other available types, and change the value of the `const` to whatever you like (here we've set the value to `42`).

In this example you'll notice an `(export "getAnswerPlus1")` section, declared just after the `func` statement in the second function — this is a shorthand way of declaring that we want to export this function, and defining the name we want to export it as.

This is functionally equivalent to including a separate function statement outside the function, elsewhere in the module in the same manner as we did before, e.g.:

WASM



```
(export "getAnswerPlus1" (func $functionName))
```

The JavaScript code to call our above module looks like so:

JS



```
WebAssembly.instantiateStreaming(fetch("call.wasm")).then((obj) => {  
  console.log(obj.instance.exports.getAnswerPlus1()); // "43"  
});
```

Importing functions from JavaScript

We have already seen JavaScript calling WebAssembly functions, but what about WebAssembly calling JavaScript functions? WebAssembly doesn't actually have any built-in knowledge of JavaScript, but it does have a general way to import functions that can accept either JavaScript or Wasm functions. Let's look at an example:

WASM



```
(module  
  (import "console" "log" (func $log (param i32)))  
  (func (export "logIt")  
    i32.const 13  
    call $log))
```


WebAssembly has a two-level namespace so the import statement here is saying that we're asking to import the `log` function from the `console` module. You can also see that the exported `logIt` function calls the imported function using the `call` instruction we introduced above.

Imported functions are just like normal functions: they have a signature that WebAssembly validation checks statically, and they are given an index and can be named and called.



JavaScript functions have no notion of signature, so any JavaScript function can be passed, regardless of the import's declared signature. Once a module declares an import, the caller of `WebAssembly.instantiate()` must pass in an import object that has the corresponding properties.

For the above, we need an object (let's call it `importObject`) such that `importObject.console.log` is a JavaScript function.

This would look like the following:



```
JS 
const importObject = {
  console: {
    log(arg) {
      console.log(arg);
    },
  },
};

WebAssembly.instantiateStreaming(fetch("logger.wasm"), importObject).then(
  (obj) => {
    obj.instance.exports.logIt();
  },
);
```

Note: You can find this example on GitHub as [logger.html](#)  (see it live [also](#) .

Declaring globals in WebAssembly

WebAssembly has the ability to create global variable instances, accessible from both JavaScript and importable/exportable across one or more [WebAssembly.Module](#) instances. This is very useful, as it allows dynamic linking of multiple modules.

In WebAssembly text format, it looks something like this (see [global.wat](#)  in our GitHub repo; also see [global.html](#)  for a live JavaScript example):

```
WASM   
(module  
  (global $g (import "js" "global") (mut i32))  
  (func (export "getGlobal") (result i32)  
    (global.get $g))  
  (func (export "incGlobal")  
    (global.set $g  
      (i32.add (global.get $g) (i32.const 1))))  
)
```

This looks similar to what we've seen before, except that we specify a global value using the keyword `global`, and we also specify the keyword `mut` along with the value's datatype if we want it to be mutable.

To create an equivalent value using JavaScript, you'd use the [WebAssembly.Global\(\)](#) constructor:

```
JS   
const global = new WebAssembly.Global({ value: "i32", mutable: true }, 0);
```

WebAssembly Memory

The examples above show how to work with numbers in assembly code, adding them to the [stack](#), perform operations on them, and then logging the result by calling a method in JavaScript.

For working with strings and other more complex data types we use `memory`, which can be created in either the WebAssembly or JavaScript, and shared

between environments (more recent versions of WebAssembly can also use [Reference types](#)).

In WebAssembly, `memory` is just a large contiguous, mutable array of raw bytes, that can grow over time (see [linear memory](#) [↗] in the specification). WebAssembly contains [memory instructions](#) like `i32.load` and `i32.store` for reading and writing bytes between the stack and any location in a memory.

From JavaScript's point of view, it's as though memory is all inside one big growable `ArrayBuffer`. JavaScript can create WebAssembly linear memory instances via the `WebAssembly.Memory()` interface and export them to a memory instance, or access a memory instance created within the WebAssembly code and exported. JavaScript `Memory` instances have a `buffer` getter, which returns an `ArrayBuffer` that points at the whole linear memory.

Memory instances can also grow, for example via the `Memory.grow()` method in JavaScript or `memory.grow` in the WebAssembly. Since `ArrayBuffer` objects can't change size, the current `ArrayBuffer` is detached and a new `ArrayBuffer` is created to point to the newer, bigger memory.

Note that when you create the memory you need to define the initial size, and you can optionally specify the maximum size to which the memory can grow. WebAssembly will attempt to reserve the maximum size (if specified), and if it is able to do so, it can grow the buffer more efficiently in future. Even if it can't allocate the maximum size now, it may still be able to grow later. The method will only fail if it cannot allocate the *initial* size.

i **Note:** Originally WebAssembly only allowed one memory per module instance. You can now have [multiple memories](#) when supported by the browser. Code that doesn't use multiple memories does not need to change!

To demonstrate some of this behaviour, let's consider the case where we want to work with a string in our WebAssembly code. A string is just a sequence of bytes somewhere inside this linear memory. Assuming we've written a suitable string of bytes to WebAssembly memory, we can pass that string to JavaScript by sharing

the memory, the offset of the string within the memory, and some way of indicating the length.

Firstly let's create some memory and share it between the WebAssembly and JavaScript. WebAssembly gives us a lot of flexibility here: we can either create a `Memory` object in JavaScript and have the WebAssembly module import the memory, or we can have the WebAssembly module create the memory and export it to JavaScript.

For this example we'll create the memory in JavaScript then import it into WebAssembly. First we create a `Memory` object with 1 page, and add it to our `importObject` under the key `js.mem`. We then instantiate our web assembly module, in this case "the_wasm_to_import.wasm", using the `WebAssembly.instantiateStreaming()` method and passing the import object:

```
JS
const memory = new WebAssembly.Memory({ initial: 1 });

const importObject = {
  js: { mem: memory },
};

WebAssembly.instantiateStreaming(
  fetch("the_wasm_to_import.wasm"),
  importObject,
).then((obj) => {
  // Call exported functions ...
});
```

Within our WebAssembly file we import this memory. Using the WebAssembly text format, the `import` statement is written as follows:

```
WASM
(import "js" "mem" (memory 1))
```

The memory must be imported using the same two-level key specified in the `importObject` (`js.mem`). The `1` indicates that the imported memory must have at least 1 page of memory (WebAssembly currently defines a page to be 64KB).

Note: As this is the first memory imported into the WebAssembly module it has a memory index of "0". You could reference this particular memory using the index in memory instructions, but since 0 is the default index, in single-memory applications you don't need to.

Now that we have a shared memory instance, the next step is to write a string of data into it. We'll then pass information about where the string is located and its length to the JavaScript (we could alternatively encode the string's length in the string itself, but passing a length is easier for us to implement).

First let's add a string of data to our memory, in this case "Hi". Since we own the entire linear memory, we can just write the string contents into global memory using a `data` section. Data sections allow a string of bytes to be written at a given offset at instantiation time and are similar to the `.data` sections in native executable formats. Here we're writing the data to the default memory (which we do not need to specify) at offset 0:

WASM



```
(module
  (import "js" "mem" (memory 1))
  ;; ...
  (data (i32.const 0) "Hi")
  ;;
)
```

Note: The double semicolon syntax (`;;`) above is used to indicate comments in WebAssembly files. In this case we're just using them to indicate placeholders for other code.

To share this data with JavaScript we'll define two functions. First we import a function from the JavaScript, which we will use to log the string to the console. This will need to be mapped to `console.log` in the `importObject` used to instantiate the WebAssembly module. The function is named `$log` in the WebAssembly and takes `i32` parameters for the string offset and length in the memory.

The second WebAssembly function, `writeHi()`, calls the imported `$log` function with the offset and length of the string in memory (`0` and `2`). This is exported from the module so that it can be called from JavaScript.

Our final WebAssembly module (in text format) looks like this.

```
WASM   
(module  
  (import "console" "log" (func $log (param i32 i32)))  
  (import "js" "mem" (memory 1))  
  (data (i32.const 0) "Hi")  
  (func (export "writeHi")  
    i32.const 0 ;; pass offset 0 to log  
    i32.const 2 ;; pass length 2 to log  
    call $log  
  )  
)
```




On the JavaScript-side we need to define the logging function, pass it to the WebAssembly, and then call the exported `writeHi()` method. The complete code is shown below:

```
JS   
const memory = new WebAssembly.Memory({ initial: 1 });  
  
// Logging function ($log) called from WebAssembly  
function consoleLogString(offset, length) {  
  const bytes = new Uint8Array(memory.buffer, offset, length);  
  const string = new TextDecoder("utf8").decode(bytes);  
  console.log(string);  
}  
  
const importObject = {  
  console: { log: consoleLogString },  
  js: { mem: memory },  
};  
  
WebAssembly.instantiateStreaming(fetch("logger2.wasm"), importObject).then(  
  (obj) => {  
    // Call the function exported from logger2.wasm  
    obj.instance.exports.writeHi();  
  })
```

```
},  
);
```

Note that the logging function `consoleLogString()` is passed to the `importObject` in the property `console.log`, and imported by the WebAssembly module. The function creates a view on the string in the shared memory using an `Uint8Array` at the passed offset and with the given length. The bytes are then decoded from UTF-8 into a string using the [TextDecoder API](#) (we specify `utf8` here, but many other encodings are supported). The string is then logged to console with `console.log()`.

The final step is to call the exported `writeHi()` function, which is done after the object is instantiated. When you run the code, the console will show the text "Hi".

 **Note:** You can find the full source on GitHub as [logger2.html](#)  (also see it [live](#) .

Multiple memories

More recent implementations allow you to use multiple memory objects in your WebAssembly and JavaScript, in a way that is compatible with code that is written for implementations that only support a single memory. Multiple memories can be useful for separating data that should be treated differently to other application data, such as public vs private data, data that needs to be persisted, and data that needs to be shared between threads. It may also be useful for very large applications that need to scale beyond the Wasm 32-bit address space, and for other purposes.

Memories that are made available to the WebAssembly code, either declared directly or imported, are given a zero-indexed sequentially allocated memory index number. All the [memory instructions](#), such as `load` or `store`, can reference any particular memory via its index so you can control which memory you're working with.


The memory instructions have a default index of 0, the index of the first memory added to the WebAssembly instance. As a result, if you only add one memory,

your code doesn't have to specify the index.

To show how this works in more detail, we'll extend the previous example to write strings to three different memories and log the results. The code below shows how we first import two memory instances, using the same approach as in the previous example. To show how you can create memory within the WebAssembly module, we've created a third memory instance named `$mem2` in the module and *exported* it.

```
WASM   
(module  
  ;; ...  
  
  (import "js" "mem0" (memory 1))  
  (import "js" "mem1" (memory 1))  
  
  ;; Create and export a third memory  
  (memory $mem2 1)  
  (export "memory2" (memory $mem2))  
  
  ;; ...  
)
```

The three memory instances are automatically assigned an instance based on their order of creation. The code below shows how we can specify this index (e.g. `(memory 1)`) in the `data` instruction to choose the memory we want to write a string to (you can use the same approach for all other memory instructions, such as `load` and `grow`). Here we write a string that indicates each memory type.

```
WASM   
  
(data (memory 0) (i32.const 0) "Memory 0 data")  
(data (memory 1) (i32.const 0) "Memory 1 data")  
(data (memory 2) (i32.const 0) "Memory 2 data")  
  
;; Add text to default (0-index) memory  
(data (i32.const 13) " (Default)")
```

Note that the `(memory 0)` is the default, and hence optional. To demonstrate this we write the text `" (Default)"` without specifying the memory index, and this

should be appended after `"Memory 0 data"` when the memory contents are logged.

The WebAssembly logging code is almost exactly the same as the previous example, except that along with the string offset and length, we need to pass the index of the memory that contains the string. We also log all three memory instances.

The complete module is shown below:

WASM 

```
(module
  (import "console" "log" (func $log (param i32 i32 i32)))

  (import "js" "mem0" (memory 1))
  (import "js" "mem1" (memory 1))

  ;; Create and export a third memory
  (memory $mem2 1)
  (export "memory2" (memory $mem2))

  (data (memory 0) (i32.const 0) "Memory 0 data")
  (data (memory 1) (i32.const 0) "Memory 1 data")
  (data (memory 2) (i32.const 0) "Memory 2 data")

  ;; Add text to default (0-index) memory
  (data (i32.const 13) " (Default)")

  (func $logMemory (param $memIndex i32) (param $memOffset i32) (param
$stringLength i32)
    local.get $memIndex
    local.get $memOffset
    local.get $stringLength
    call $log
  )

  (func (export "logAllMemory")
    ;; Log memory index 0, offset 0
    (i32.const 0) ;; memory index 0
    (i32.const 0) ;; memory offset 0
    (i32.const 23) ;; string length 23
    (call $logMemory)
```


```

;; Log memory index 1, offset 0
i32.const 1 ;; memory index 1
i32.const 0 ;; memory offset 0
i32.const 20 ;; string length 20
call $logMemory

;; Log memory index 2, offset 0
i32.const 2 ;; memory index 2
i32.const 0 ;; memory offset 0
i32.const 12 ;; string length 13
call $logMemory
)
)

```

The JavaScript code is also very similar to the previous example, except that we create and pass two memory instances to the `importObject()` and the memory exported by the module instance is accessed after it has been instantiated using the resolved promise (`obj.instance.exports`). The code to log each string is also a little more complicated because we need to match the memory instance number from the WebAssembly to a particular `Memory` object.

JS 

```

const memory0 = new WebAssembly.Memory({ initial: 1 });
const memory1 = new WebAssembly.Memory({ initial: 1 });
let memory2; // Created by module

function consoleLogString(memoryInstance, offset, length) {
  let memory;
  switch (memoryInstance) {
    case 0:
      memory = memory0;
      break;
    case 1:
      memory = memory1;
      break;
    case 2:
      memory = memory2;
      break;
  }
  // code block
}

```

```

const bytes = new Uint8Array(memory.buffer, offset, length);
const string = new TextDecoder("utf8").decode(bytes);
log(string); // implementation not shown – could call console.log()
}

const importObject = {
  console: { log: consoleLogString },
  js: { mem0: memory0, mem1: memory1 },
};

WebAssembly.instantiateStreaming(fetch("multi-memory.wasm"),
importObject).then(
  (obj) => {
    //Get exported memory
    memory2 = obj.instance.exports.memory2;
    //Log memory
    obj.instance.exports.logAllMemory();
  },
);

```


The output of the example should be similar to the text below, except that "Memory 1 data" may have some trailing "rubbish characters", because the text decoder is passed more bytes than are used to encode the string.

```

Memory 0 data (Default)
Memory 1 data
Memory 2 data

```

You can find the full source on GitHub as [multi-memory.html](#)  ([also see it live](#) )

 **Note:** See [webassembly.multiMemory](#) in the [home page](#) for browser compatibility information for this feature.

WebAssembly tables

To finish this tour of the WebAssembly text format, let's look at the most intricate, and often confusing, part of WebAssembly: **tables**. Tables are basically resizable arrays of references that can be accessed by index from WebAssembly code.

To see why tables are needed, we need to first observe that the `call` instruction we saw earlier (see [Calling functions from other functions in the same module](#)) takes a static function index and thus can only ever call one function — but what if the callee is a runtime value?

- In JavaScript, we see this all the time: functions are first-class values.
- In C/C++, we see this with function pointers.
- In C++, we see this with virtual functions.

WebAssembly needed a type of call instruction to achieve this, so we gave it `call_indirect`, which takes a dynamic function operand. The problem is that the only types we have to give operands in WebAssembly are (currently) `i32 / i64 / f32 / f64`.

WebAssembly could add an `anyfunc` type ("any" because the type could hold functions of any signature), but unfortunately this `anyfunc` type couldn't be stored in linear memory for security reasons. Linear memory exposes the raw contents of stored values as bytes and this would allow Wasm content to arbitrarily observe and corrupt raw function addresses, which is something that cannot be allowed on the web.

The solution was to store function references in a table and pass around table indices instead, which are just `i32` values. `call_indirect`'s operand can therefore be an `i32` index value.

Defining a table in Wasm

So how do we place Wasm functions in our table? Just like `data` sections can be used to initialize regions of linear memory with bytes, `elem` sections can be used to initialize regions of tables with functions:

WASM



```
(module
  (table 2 funcref)
  (elem (i32.const 0) $f1 $f2)
  (func $f1 (result i32)
    i32.const 42)
```

```
(func $f2 (result i32)
  i32.const 13)
...
)
```

- In `(table 2 funcref)`, the 2 is the initial size of the table (meaning it will store two references) and `funcref` declares that the element type of these references are function reference.
- The functions `(func)` sections are just like any other declared Wasm functions. These are the functions we are going to refer to in our table (for example's sake, each one just returns a constant value). Note that the order the sections are declared in doesn't matter here — you can declare your functions anywhere and still refer to them in your `elem` section.
- The `elem` section can list any subset of the functions in a module, in any order, allowing duplicates. This is a list of the functions that are to be referenced by the table, in the order they are to be referenced.
- The `(i32.const 0)` value inside the `elem` section is an offset — this needs to be declared at the start of the section, and specifies at what index in the table function references start to be populated. Here we've specified 0, and a size of 2 (see above), so we can fill in two references at indexes 0 and 1. If we wanted to start writing our references at offset 1, we'd have to write `(i32.const 1)`, and the table size would have to be 3.

Note: Uninitialized elements are given a default throw-on-call value.

In JavaScript, the equivalent calls to create such a table instance would look something like this:


```
JS
function () {
  // table section
  const tbl = new WebAssembly.Table({initial: 2, element: "anyfunc"});

  // function sections:
  const f1 = ... /* some imported WebAssembly function */
  const f2 = ... /* some imported WebAssembly function */
}
```

```
// elem section
tbl.set(0, f1);
tbl.set(1, f2);
};
```

Using the table

Moving on, now we've defined the table we need to use it somehow. Let's use this section of code to do so:

```
WASM 
(type $return_i32 (func (result i32))) ;; if this was f32, type checking
would fail
(func (export "callByIndex") (param $i i32) (result i32)
  local.get $i
  call_indirect (type $return_i32))
```

- The `(type $return_i32 (func (result i32)))` block specifies a type, with a reference name. This type is used when performing type checking of the table function reference calls later on. Here we are saying that the references need to be functions that return an `i32` as a result.
- Next, we define a function that will be exported with the name `callByIndex`. This will take one `i32` as a parameter, which is given the argument name `$i`.
- Inside the function, we add one value to the stack — whatever value is passed in as the parameter `$i`.
- Finally, we use `call_indirect` to call a function from the table — it implicitly pops the value of `$i` off the stack. The net result of this is that the `callByIndex` function invokes the `$i`'th function in the table.

You could also declare the `call_indirect` parameter explicitly during the command call instead of before it, like this:

```
WASM 
(call_indirect (type $return_i32) (local.get $i))
```

In a higher level, more expressive language like JavaScript, you could imagine doing the same thing with an array (or probably more likely, object) containing

functions. The pseudocode would look something like `tbl[i]()`.

So, back to the typechecking. Since WebAssembly is type checked, and the `funcref` can be potentially any function signature, we have to supply the presumed signature of the callee at the callsite, hence we include the `$return_i32` type, to tell the program a function returning an `i32` is expected. If the callee doesn't have a matching signature (say an `f32` is returned instead), a [WebAssembly.RuntimeError](#) is thrown.

So what links the `call_indirect` to the table we are calling? The answer is that there is only one table allowed right now per module instance, and that is what `call_indirect` is implicitly calling. In the future, when multiple tables are allowed, we would also need to specify a table identifier of some kind, along the lines of

```
WASM
call_indirect $my_spicy_table (type $i32_to_void)
```

The full module all together looks like this, and can be found in our [wasm-table.wat](#) [example file](#):

```
WASM
(module
  (table 2 funcref)
  (func $f1 (result i32)
    i32.const 42)
  (func $f2 (result i32)
    i32.const 13)
  (elem (i32.const 0) $f1 $f2)
  (type $return_i32 (func (result i32)))
  (func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32))
)
```

We load it into a webpage using the following JavaScript:

```
JS
```

```
WebAssembly.instantiateStreaming(fetch("wasm-table.wasm")).then((obj) => {
  console.log(obj.instance.exports.callByIndex(0)); // returns 42
  console.log(obj.instance.exports.callByIndex(1)); // returns 13
  console.log(obj.instance.exports.callByIndex(2)); // returns an error,
  because there is no index position 2 in the table
});
```

Note: You can find this example on GitHub as [wasm-table.html](#) (see it [live also](#)).

Note: Just like Memory, Tables can also be created from JavaScript (see [WebAssembly.Table\(\)](#)) as well as imported to/from another Wasm module.

Mutating tables and dynamic linking

Because JavaScript has full access to function references, the Table object can be mutated from JavaScript using the [grow\(\)](#), [get\(\)](#) and [set\(\)](#) methods. And WebAssembly code is itself able to manipulate tables using instructions added as part of [Reference types](#), such as `table.get` and `table.set`.

Because tables are mutable, they can be used to implement sophisticated load-time and run-time [dynamic linking schemes](#). When a program is dynamically linked, multiple instances share the same memory and table. This is symmetric to a native application where multiple compiled `.dll`s share a single process's address space.

To see this in action, we'll create a single import object containing a Memory object and a Table object, and pass this same import object to multiple [instantiate\(\)](#) calls.

Our `.wat` examples look like so:

```
shared0.wat :
```



```
(module
  (import "js" "memory" (memory 1))
  (import "js" "table" (table 1 funcref))
  (elem (i32.const 0) $shared0func)
  (func $shared0func (result i32)
    i32.const 0
    i32.load)
)
```

shared1.wat :

WASM



```
(module
  (import "js" "memory" (memory 1))
  (import "js" "table" (table 1 funcref))
  (type $void_to_i32 (func (result i32)))
  (func (export "doIt") (result i32)
    i32.const 0
    i32.const 42
    i32.store ;; store 42 at address 0
    i32.const 0
    call_indirect (type $void_to_i32))
)
```

These work as follows:

1. The function `shared0func` is defined in `shared0.wat`, and stored in our imported table.
2. This function creates a constant containing the value `0`, and then uses the `i32.load` command to load the value contained in the provided memory index. The index provided is `0` — again, it implicitly pops the previous value off the stack. So `shared0func` loads and returns the value stored at memory index `0`.
3. In `shared1.wat`, we export a function called `doIt` — this function creates two constants containing the values `0` and `42`, then calls `i32.store` to store a provided value at a provided index of the imported memory. Again, it implicitly pops these values off the stack, so the result is that it stores the value `42` in memory index `0`,

4. In the last part of the function, we create a constant with value `0`, then call the function at this index 0 of the table, which is `shared0func`, stored there earlier by the `elem` block in `shared0.wat`.
5. When called, `shared0func` loads the `42` we stored in memory using the `i32.store` command in `shared1.wat`.

Note: The above expressions again pop values from the stack implicitly, but you could declare these explicitly inside the command calls instead, for example:

```
WASM
(i32.store (i32.const 0) (i32.const 42))
(call_indirect (type $void_to_i32) (i32.const 0))
```

After converting to assembly, we then use `shared0.wasm` and `shared1.wasm` in JavaScript via the following code:

```
JS
const importObj = {
  js: {
    memory: new WebAssembly.Memory({ initial: 1 }),
    table: new WebAssembly.Table({ initial: 1, element: "anyfunc" }),
  },
};

Promise.all([
  WebAssembly.instantiateStreaming(fetch("shared0.wasm"), importObj),
  WebAssembly.instantiateStreaming(fetch("shared1.wasm"), importObj),
]).then((results) => {
  console.log(results[1].instance.exports.doIt()); // prints 42
});
```

Each of the modules that is being compiled can import the same memory and table objects and thus share the same linear memory and table "address space".

Note: You can find this example on GitHub as [shared-address-](#)

[space.html](#) [↗](#) (see it live also [↗](#)).

Bulk memory operations

Bulk memory operations are a newer addition to the language — seven new built-in operations are provided for bulk memory operations such as copying and initializing, to allow WebAssembly to model native functions such as `memcpy` and `memmove` in a more efficient, performant way.

Note: See [webassembly.bulk-memory-operations](#) in the home page for browser compatibility information.

The new operations are:

- `data.drop`: Discard the data in an data segment.
- `elem.drop`: Discard the data in an element segment.
- `memory.copy`: Copy from one region of linear memory to another.
- `memory.fill`: Fill a region of linear memory with a given byte value.
- `memory.init`: Copy a region from a data segment.
- `table.copy`: Copy from one region of a table to another.
- `table.init`: Copy a region from an element segment.

Note: You can find more information in the [Bulk Memory Operations and Conditional Segment Initialization](#) [↗](#) proposal.

Types

Number types

WebAssembly currently has four available *number types*:

- `i32`: 32-bit integer

- `i64` : 64-bit integer
- `f32` : 32-bit float
- `f64` : 64-bit float

Vector types

- `v128` : 128 bit vector of packed integer, floating-point data, or a single 128 bit type.

Reference types

The [reference types proposal](#) [↗] provides two main features:

- A new type, `externref`, which can hold *any* JavaScript value, for example strings, DOM references, objects, etc. `externref` is opaque from the point of view of WebAssembly — a Wasm module can't access and manipulate these values and instead can only receive them and pass them back out. But this is very useful for allowing Wasm modules to call JavaScript functions, DOM APIs, etc., and generally to pave the way for easier interoperability with the host environment. `externref` can be used for value types and table elements.
- A number of new instructions that allow Wasm modules to directly manipulate [WebAssembly tables](#), rather than having to do it via the JavaScript API.

i **Note:** The [wasm-bindgen](#) [↗] documentation contains some useful information on how to take advantage of `externref` from Rust.

i **Note:** See [webassembly.reference-types](#) in the [home page](#) for browser compatibility information.


Multi-value WebAssembly

Another more recent addition to the language is WebAssembly multi-value, meaning that WebAssembly functions can now return multiple values, and instruction sequences can consume and produce multiple stack values.

Note: See [webassembly.multi-value](#) in the home page for browser compatibility information.

At the time of writing (June 2020) this is at an early stage, and the only multi-value instructions available are calls to functions that themselves return multiple values. For example:

```
WASM
(module
  (func $get_two_numbers (result i32 i32)
    i32.const 1
    i32.const 2
  )
  (func (export "add_two_numbers") (result i32)
    call $get_two_numbers
    i32.add
  )
)
```

But this will pave the way for more useful instruction types, and other things besides. For a useful write-up of progress so far and how this works, see [Multi-Value All The Wasm!](#)  by Nick Fitzgerald.

WebAssembly threads

WebAssembly Threads allow WebAssembly Memory objects to be shared across multiple WebAssembly instances running in separate Web Workers, in the same fashion as [SharedArrayBuffer](#) in JavaScript. This allows very fast communication between Workers, and significant performance gains in web applications.

The threads proposal has two parts, shared memories and atomic memory accesses.

Note: See [webassembly.threads-and-atomics](#) in the home page for browser compatibility information.

Shared memories

As described above, you can create shared WebAssembly `Memory` objects, which can be transferred between Window and Worker contexts using `postMessage()`, in the same fashion as a `SharedArrayBuffer`.

Over on the JavaScript API side, the `WebAssembly.Memory()` constructor's initialization object now has a `shared` property, which when set to `true` will create a shared memory:

```
JS
const memory = new WebAssembly.Memory({
  initial: 10,
  maximum: 100,
  shared: true,
});
```

the memory's `buffer` property will now return a `SharedArrayBuffer`, instead of the usual `ArrayBuffer`:

```
JS
memory.buffer; // returns SharedArrayBuffer
```


Over in the text format, you can create a shared memory using the `shared` keyword, like this:



```
WASM
(memory 1 2 shared)
```

Unlike unshared memories, shared memories must specify a "maximum" size, in both the JavaScript API constructor and Wasm text format.

Note: You can find a lot more details in the [Threading proposal for WebAssembly](#).

Atomic memory accesses



A number of new Wasm instructions have been added that can be used to implement higher level features like mutexes, condition variables, etc. You can [find them listed here](#) .

 **Note:** The [Emscripten Pthreads support page](#)  shows how to take advantage of this new functionality from Emscripten.

Summary

This finishes our high-level tour of the major components of the WebAssembly text format and how they get reflected in the WebAssembly JS API.

See also

- The main thing that wasn't included is a comprehensive list of all the instructions that can occur in function bodies. See the [WebAssembly semantics](#)  for a treatment of each instruction.
- See also the [grammar of the text format](#)  that is implemented by the spec interpreter.

Help improve MDN

Was this page helpful to you?



[Learn how to contribute.](#)

This page was last modified on Oct 26, 2024 by [MDN contributors](#).

