disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2019 W3C Process Document.

Table of Contents

This API provides a way to access WebAssembly [WEBASSEMBLY] through a bridge to explicitly construct modules from JavaScript [ECMASCRIPT].

§ 1. Sample API Usage

This section is non-normative.

Given `demo.wat` (encoded to `demo.wasm`):

```
(module
    (import "js" "import1" (func $i1))
    (import "js" "import2" (func $i2))
    (func $main (call $i1))
    (start $main)
    (func (export "f") (call $i2))
)
```

and the following JavaScript, run in a browser:

```
var importObj = {js: {
    import1: () => console.log("hello,"),
    import2: () => console.log("world!")
}};
fetch('demo.wasm').then(response =>
    response.arrayBuffer()
).then(buffer =>
    WebAssembly.instantiate(buffer, importObj)
).then(({module, instance}) =>
    instance.exports.f()
);
```

§ 2. Notation

This specification depends on the Infra Standard. [INFRA]

The WebAssembly sequence type is equivalent to the list type defined there; values of one are treated as values of the other transparently.

§ 3. Internal storage

§ 3.1. Interaction of the WebAssembly Store with JavaScript

> Note: WebAssembly semantics are defined in terms of an abstract store, representing the state of the WebAssembly abstract machine. WebAssembly operations take a store and return an updated store.

Each agent has an ***associated store***. When a new agent is created, its associated store is set to the result of store_init().

> Note: In this specification, no WebAssembly-related objects, memory or addresses can be shared among agents in an agent cluster. In a future version of WebAssembly, this may change.

Elements of the WebAssembly store may be ***identified with*** JavaScript values. In particular, each WebAssembly memory instance with a corresponding `Memory` object is identified with a JavaScript Data Block; modifications to this Data Block are identified to updating the agent's store to a store which reflects those changes, and vice versa.

§ 3.2. WebAssembly JS Object Caches

> Note: There are several WebAssembly objects that may have a corresponding JavaScript object. The correspondence is stored in a per-agent mapping from WebAssembly addresses to JavaScript objects. This mapping is used to ensure that, for a given agent, there exists at most one JavaScript object for a particular WebAssembly address. However, this property does not hold for shared objects.

Each agent is associated with the following ordered maps:

- The ***Memory object cache***, mapping memory addresses to `Memory` objects.
- The ***Table object cache***, mapping table addresses to `Table` objects.
- The ***Exported Function cache***, mapping function addresses to Exported Function objects.
- The ***Global object cache***, mapping global addresses to `Global` objects.

§ 4. The WebAssembly Namespace

```
dictionary WebAssemblyInstantiatedSource {
    required Module module;
    required Instance instance;
};

[Exposed=(Window,Worker,Worklet)]
namespace WebAssembly {
    boolean validate(BufferSource bytes);
    Promise<Module> compile(BufferSource bytes);

    Promise<WebAssemblyInstantiatedSource> instantiate(
        BufferSource bytes, optional object importObject);

    Promise<Instance> instantiate(
        Module moduleObject, optional object importObject);
};
```

To ***compile a WebAssembly module*** from source bytes *bytes*, perform the following steps:

1. Let *module* be module_decode(*bytes*). If *module* is error, return error.
2. If module_validate(*module*) is error, return error.

3. Return *module*.

The ***validate(*bytes*)*** method, when invoked, performs the following steps:

1. Let *stableBytes* be a copy of the bytes held by the buffer *bytes*.
2. Compile *stableBytes* as a WebAssembly module and store the results as *module*.
3. If *module* is error, return false.
4. Return true.

A `Module` object represents a single WebAssembly module. Each `Module` object has the following internal slots:

- [[Module]] : a WebAssembly module
- [[Bytes]] : the source bytes of [[Module]].

To ***construct a WebAssembly module object*** from a module *module* and source bytes *bytes*, perform the following steps:

1. Let *moduleObject* be a new `Module` object.
2. Set *moduleObject*.[[Module]] to *module*.
3. Set *moduleObject*.[[Bytes]] to *bytes*.
4. Return *moduleObject*.

To ***asynchronously compile a WebAssembly module*** from source bytes *bytes*, using optional task source *taskSource*, perform the following steps:

1. Let *promise* be a new promise.
2. Run the following steps in parallel:
    1. Compile the WebAssembly module *bytes* and store the result as *module*.
    2. Queue a task to perform the following steps. If *taskSource* was provided, queue the task on that task source.
        1. If *module* is error, reject *promise* with a `CompileError` exception.
        2. Otherwise,
            1. Construct a WebAssembly module object from *module* and *bytes*, and let *moduleObject* be the result.
            2. Resolve *promise* with *moduleObject*.
3. Return *promise*.

The ***compile(*bytes*)*** method, when invoked, performs the following steps:

1. Let *stableBytes* be a copy of the bytes held by the buffer *bytes*.
2. Asynchronously compile a WebAssembly module from *stableBytes* and return the result.

To ***read the imports*** from a WebAssembly module *module* from imports object *importObject*, perform the following steps:

1. If *module*.imports is not empty, and *importObject* is undefined, throw a `TypeError` exception.

2. Let *imports* be « ».
3. For each (*moduleName*, *componentName*, *externtype*) of module_imports(*module*),
    1. Let *o* be ? Get(*importObject*, *moduleName*).
    2. If Type(*o*) is not Object, throw a `TypeError` exception.
    3. Let *v* be ? Get(*o*, *componentName*).
    4. If *externtype* is of the form func *functype*,
        1. If IsCallable(*v*) is false, throw a `LinkError` exception.
        2. If *v* has a [[FunctionAddress]] internal slot, and therefore is an Exported Function,
            1. Let *funcaddr* be the value of *v*'s [[FunctionAddress]] internal slot.
        3. Otherwise,
            1. Create a host function from *v* and *functype*, and let *funcaddr* be the result.
            2. Let *index* be the number of external functions in *imports*. This value *index* is known as the ***index of the host function*** *funcaddr*.
        4. Let *externfunc* be the external value func *funcaddr*.
        5. Append *externfunc* to *imports*.
    5. If *externtype* is of the form global *mut valtype*,
        1. If Type(*v*) is Number or BigInt,
            1. If *valtype* is i64 and Type(*v*) is Number,
                1. Throw a `LinkError` exception.
            2. If *valtype* is not i64 and Type(*v*) is BigInt,
                1. Throw a `LinkError` exception.
            3. Let *value* be ToWebAssemblyValue(*v*, *valtype*).
            4. Let *store* be the surrounding agent's associated store.
            5. Let (*store*, *globaladdr*) be global_alloc(*store*, const *valtype*, *value*).
            6. Set the surrounding agent's associated store to *store*.
        2. Otherwise, if *v* implements `Global`,
            1. Let *globaladdr* be *v*.[[Global]].
        3. Otherwise,
            1. Throw a `LinkError` exception.
        4. Let *externglobal* be global *globaladdr*.
        5. Append *externglobal* to *imports*.
    6. If *externtype* is of the form mem *memtype*,

1. If *v* does not implement `Memory`, throw a `LinkError` exception.
2. Let *externmem* be the external value mem *v*.[[Memory]].
3. Append *externmem* to *imports*.

7. If *externtype* is of the form table *tabletype*,

1. If *v* does not implement `Table`, throw a `LinkError` exception.
2. Let *tableaddr* be *v*.[[Table]].
3. Let *externtable* be the external value table *tableaddr*.
4. Append *externtable* to *imports*.

4. Return *imports*.

Note: This algorithm only verifies the right kind of JavaScript values are passed. The verification of WebAssembly type requirements is deferred to the "instantiate the core of a WebAssembly module" algorithm.

To ***create an exports object*** from a WebAssembly module *module* and instance *instance*, perform the following steps:

1. Let *exportsObject* be ! ObjectCreate(null).
2. For each (*name*, *externtype*) of module_exports(*module*),

1. Let *externval* be instance_export(*instance*, *name*).
2. Assert: *externval* is not error.
3. If *externtype* is of the form func *functype*,

1. Assert: *externval* is of the form func *funcaddr*.
2. Let func *funcaddr* be *externval*.
3. Let *func* be the result of creating a new Exported Function from *funcaddr*.
4. Let *value* be *func*.

4. If *externtype* is of the form global *mut globaltype*,

1. Assert: *externval* is of the form global *globaladdr*.
2. Let global *globaladdr* be *externval*.
3. Let *global* be a new Global object created from *globaladdr*.
4. Let *value* be *global*.

5. If *externtype* is of the form mem *memtype*,

1. Assert: *externval* is of the form mem *memaddr*.
2. Let mem *memaddr* be *externval*.
3. Let *memory* be a new Memory object created from *memaddr*.
4. Let *value* be *memory*.

6. If *externtype* is of the form table *tabletype*,

1. Assert: *externval* is of the form table *tableaddr*.
2. Let table *tableaddr* be *externval*.
3. Let *table* be a new Table object created from *tableaddr*.
4. Let *value* be *table*.

7. Let *status* be ! CreateDataProperty(*exportsObject*, *name*, *value*).
8. Assert: *status* is true.

Note: the validity and uniqueness checks performed during WebAssembly module validation ensure that each property name is valid and no properties are defined twice.

3. Perform ! SetIntegrityLevel(*exportsObject*, `"frozen"`).
4. Return *exportsObject*.

To ***initialize an instance object*** *instanceObject* from a WebAssembly module *module* and instance *instance*, perform the following steps:

1. Create an exports object from *module* and *instance* and let *exportsObject* be the result.
2. Set *instanceObject*.[[Instance]] to *instance*.
3. Set *instanceObject*.[[Exports]] to *exportsObject*.

To ***instantiate the core of a WebAssembly module*** from a module *module* and imports *imports*, perform the following steps:

1. Let *store* be the surrounding agent's associated store.
2. Let *result* be module_instantiate(*store*, *module*, *imports*).
3. If *result* is error, throw an appropriate exception type:
   - A `LinkError` exception for most cases which occur during linking.
   - If the error came when running the start function, throw a `RuntimeError` for most errors which occur from WebAssembly, or the error object propagated from inner ECMAScript code.
   - Another error type if appropriate, for example an out-of-memory exception, as documented in the WebAssembly error mapping.
4. Let (*store*, *instance*) be *result*.
5. Set the surrounding agent's associated store to *store*.
6. Return *instance*.

To ***asynchronously instantiate a WebAssembly module*** from a `Module` *moduleObject* and imports *importObject*, perform the following steps:

1. Let *promise* be a new promise.
2. Let *module* be *moduleObject*.[[Module]].
3. Read the imports of *module* with imports *importObject*, and let *imports* be the result. If this operation throws an exception, catch it, reject *promise* with the exception, and return *promise*.

exception, catch it, reject *promise* with the exception, and return *promise*.

4. Queue a task to perform the following steps:

    1. Instantiate the core of a WebAssembly module *module* with *imports*, and let *instance* be the result. If this throws an exception, catch it, reject *promise* with the exception, and terminate these substeps.
    2. Let *instanceObject* be a new `Instance`.
    3. Initialize *instanceObject* from *module* and *instance*. If this throws an exception, catch it, reject *promise* with the exception, and terminate these substeps.
    4. Resolve *promise* with *instanceObject*.

5. Return *promise*.

To ***instantiate a WebAssembly module*** from a `Module` *moduleObject* and imports *importObject*, perform the following steps:

1. Let *module* be *moduleObject*.[[Module]].
2. Read the imports of *module* with imports *importObject*, and let *imports* be the result.
3. Instantiate the core of a WebAssembly module *module* with *imports*, and let *instance* be the result.
4. Let *instanceObject* be a new `Instance`.
5. Initialize *instanceObject* from *module* and *instance*.
6. Return *instanceObject*.

To ***instantiate a promise of a module*** *promiseOfModule* with imports *importObject*, perform the following steps:

1. Let *promise* be a new promise.
2. Upon fulfillment of *promiseOfModule* with value *module*:

    1. Instantiate the WebAssembly module *module* importing *importObject*, and let *instance* be the result. If this throws an exception, catch it, reject *promise* with the exception, and abort these substeps.
    2. Let *result* be the `WebAssemblyInstantiatedSource` value «["`module`" → *module*, "`instance`" → *instance*]».
    3. Resolve *promise* with *result*.

3. Upon rejection of *promiseOfModule* with reason *reason*:

    1. Reject *promise* with *reason*.

4. Return *promise*.

> Note: It would be valid to perform certain parts of the instantiation in parallel, but several parts need to happen in the event loop, including JavaScript operations to access the *importObject* and execution of the start function.

The ***`instantiate(bytes, importObject)`*** method, when invoked, performs the following steps:

1. Let *stableBytes* be a copy of the bytes held by the buffer *bytes*.
2. Asynchronously compile a WebAssembly module from *stableBytes* and let *promiseOfModule* be the result.

3. Instantiate *promiseOfModule* with imports *importObject* and return the result.

The ***instantiate(moduleObject, importObject)*** method, when invoked, performs the following steps:

1. Asynchronously instantiate the WebAssembly module *moduleObject* importing *importObject*, and return the result.

> Note: A follow-on streaming API is documented in the WebAssembly Web API.

§ 4.1. Modules

```
enum ImportExportKind {
  "function",
  "table",
  "memory",
  "global"
};

dictionary ModuleExportDescriptor {
  required USVString name;
  required ImportExportKind kind;
  // Note: Other fields such as signature may be added in the future.
};

dictionary ModuleImportDescriptor {
  required USVString module;
  required USVString name;
  required ImportExportKind kind;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Module {
  constructor(BufferSource bytes);
  static sequence<ModuleExportDescriptor> exports(Module moduleObject);
  static sequence<ModuleImportDescriptor> imports(Module moduleObject);
  static sequence<ArrayBuffer> customSections(Module moduleObject, DOMString sectionName
};
```

The ***string value of the extern type*** *type* is

- "function" if *type* is of the form func *functype*
- "table" if *type* is of the form table *tabletype*
- "memory" if *type* is of the form mem *memtype*
- "global" if *type* is of the form global *globaltype*

The ***exports(moduleObject)*** method, when invoked, performs the following steps:

1. Let *module* be *moduleObject*.[[Module]].

2. Let *exports* be « ».
3. For each (*name*, *type*) of module_exports(*module*),
    1. Let *kind* be the string value of the extern type *type*.
    2. Let *obj* be «["`name`" → *name*, "`kind`" → *kind*]».
    3. Append *obj* to *exports*.
4. Return *exports*.

The ***`imports(moduleObject)`*** method, when invoked, performs the following steps:

1. Let *module* be *moduleObject*.[[Module]].
2. Let *imports* be « ».
3. For each (*moduleName*, *name*, *type*) of module_imports(*module*),
    1. Let *kind* be the string value of the extern type *type*.
    2. Let *obj* be «["`module`" → *moduleName*, "`name`" → *name*, "`kind`" → *kind*]».
    3. Append *obj* to *imports*.
4. Return *imports*.

The ***`customSections(moduleObject, sectionName)`*** method, when invoked, performs the following steps:

1. Let *bytes* be *moduleObject*.[[Bytes]].
2. Let *customSections* be « ».
3. For each custom section *customSection* of *bytes*, interpreted according to the module grammar,
    1. Let *name* be the `name` of *customSection*, decoded as UTF-8.
    2. Assert: *name* is not failure (*moduleObject*.[[Module]] is valid).
    3. If *name* equals *sectionName* as string values,
        1. Append a new `ArrayBuffer` containing a copy of the bytes in *bytes* for the range matched by this customsec production to *customSections*.
4. Return *customSections*.

The ***`Module(bytes)`*** constructor, when invoked, performs the follwing steps:

1. Let *stableBytes* be a copy of the bytes held by the buffer *bytes*.
2. Compile the WebAssembly module *stableBytes* and store the result as *module*.
3. If *module* is error, throw a `CompileError` exception.
4. Set **this**.[[Module]] to *module*.
5. Set **this**.[[Bytes]] to *stableBytes*.

§ 4.2. Instances

```
[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Instance {
  constructor(Module module, optional object importObject);
  readonly attribute object exports;
};
```

The ***Instance(*module, importObject*)*** constructor, when invoked, runs the following steps:

1. Let *module* be *module*.[[Module]].
2. Read the imports of *module* with imports *importObject*, and let *imports* be the result.
3. Instantiate the core of a WebAssembly module *module* with *imports*, and let *instance* be the result.
4. Initialize **this** from *module* and *instance*.

The getter of the ***exports*** attribute of `Instance` returns **this**.[[Exports]].

§ 4.3. Memories

```
dictionary MemoryDescriptor {
  required [EnforceRange] unsigned long initial;
  [EnforceRange] unsigned long maximum;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Memory {
  constructor(MemoryDescriptor descriptor);
  unsigned long grow([EnforceRange] unsigned long delta);
  readonly attribute ArrayBuffer buffer;
};
```

A `Memory` object represents a single memory instance which can be simultaneously referenced by multiple `Instance` objects. Each `Memory` object has the following internal slots:

- [[Memory]] : a memory address
- [[BufferObject]] : an `ArrayBuffer` whose Data Block is identified with the above memory address

To ***create a memory buffer*** from a memory address *memaddr*, perform the following steps:

1. Let *block* be a Data Block which is identified with the underlying memory of *memaddr*.
2. Let *buffer* be a new `ArrayBuffer` whose [[ArrayBufferData]] is *block* and [[ArrayBufferByteLength]] is set to the length of *block*.
3. Set *buffer*.[[ArrayBufferDetachKey]] to "WebAssembly.Memory".
4. Return *buffer*.

To ***initialize a memory object*** *memory* from a memory address *memaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Memory object cache.
2. Assert: *map*[*memaddr*] doesn't exist.
3. Let *buffer* be the result of creating a memory buffer from *memaddr*.
4. Set *memory*.[[Memory]] to *memaddr*.
5. Set *memory*.[[BufferObject]] to *buffer*.
6. Set *map*[*memaddr*] to *memory*.

To ***create a memory object*** from a memory address *memaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Memory object cache.
2. If *map*[*memaddr*] exists,
    1. Return *map*[*memaddr*].
3. Let *memory* be a new `Memory`.
4. Initialize *memory* from *memaddr*.
5. Return *memory*.

The ***`Memory(descriptor)`*** constructor, when invoked, performs the following steps:

1. Let *initial* be *descriptor*["initial"].
2. If *descriptor*["maximum"] exists, let *maximum* be *descriptor*["maximum"]; otherwise, let *maximum* be empty.
3. If *maximum* is not empty and *maximum* < *initial*, throw a `RangeError` exception.
4. Let *memtype* be { min *initial*, max *maximum* }.
5. Let *store* be the surrounding agent's associated store.
6. Let (*store*, *memaddr*) be mem_alloc(*store*, *memtype*). If allocation fails, throw a `RangeError` exception.
7. Set the surrounding agent's associated store to *store*.
8. Initialize **this** from *memaddr*.

To ***reset the Memory buffer*** of *memaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Memory object cache.
2. Assert: *map*[*memaddr*] exists.
3. Let *memory* be *map*[*memaddr*].
4. Perform ! DetachArrayBuffer(*memory*.[[BufferObject]], "WebAssembly.Memory").
5. Let *buffer* be the result of creating a memory buffer from *memaddr*.
6. Set *memory*.[[BufferObject]] to *buffer*.

The ***`grow(delta)`*** method, when invoked, performs the following steps:

1. Let *store* be the surrounding agent's associated store.
2. Let *memaddr* be **this**.[[Memory]].

3. Let *ret* be the mem_size(*store*, *memaddr*).
4. Let *store* be mem_grow(*store*, *memaddr*, *delta*).
5. If *store* is error, throw a `RangeError` exception.
6. Set the surrounding agent's associated store to *store*.
7. Reset the memory buffer of *memaddr*.
8. Return *ret*.

Immediately after a WebAssembly memory.grow instruction executes, perform the following steps:

1. If the top of the stack is not i32.const (−1),

    1. Let *frame* be the current frame.
    2. Assert: due to validation, *frame*.module.memaddrs[0] exists.
    3. Let *memaddr* be the memory address *frame*.module.memaddrs[0].
    4. Reset the memory buffer of *memaddr*.

The getter of the ***buffer*** attribute of `Memory` returns **this**.[[BufferObject]].

§ 4.4. Tables

```
enum TableKind {
  "anyfunc",
  // Note: More values may be added in future iterations,
  // e.g., typed function references, typed GC references
};

dictionary TableDescriptor {
  required TableKind element;
  required [EnforceRange] unsigned long initial;
  [EnforceRange] unsigned long maximum;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Table {
  constructor(TableDescriptor descriptor);
  unsigned long grow([EnforceRange] unsigned long delta);
  Function? get([EnforceRange] unsigned long index);
  void set([EnforceRange] unsigned long index, Function? value);
  readonly attribute unsigned long length;
};
```

A `Table` object represents a single table instance which can be simultaneously referenced by multiple `Instance` objects. Each `Table` object has a [[Table]] internal slot, which is a table address.

To ***initialize a table object*** *table* from a table address *tableaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Table object cache.
2. Assert: *map*[*tableaddr*] doesn't exist.
3. Set *table*.[[Table]] to *tableaddr*.
4. Set *map*[*tableaddr*] to *table*.

To ***create a table object*** from a table address *tableaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Table object cache.
2. If *map*[*tableaddr*] exists,
    1. Return *map*[*tableaddr*].
3. Let *table* be a new `Table`.
4. Initialize *table* from *tableaddr*.
5. Return *table*.

The ***`Table(descriptor)`*** constructor, when invoked, performs the following steps:

1. Let *initial* be *descriptor*["initial"].
2. If *descriptor*["maximum"] exists, let *maximum* be *descriptor*["maximum"]; otherwise, let *maximum* be empty.
3. If *maximum* is not empty and *maximum* < *initial*, throw a `RangeError` exception.
4. Let *type* be the table type {min *initial*, max *maximum*} anyfunc.
5. Let *store* be the surrounding agent's associated store.
6. Let (*store*, *tableaddr*) be table_alloc(*store*, *type*).
7. Set the surrounding agent's associated store to *store*.
8. Initialize **this** from *tableaddr*.

The ***`grow(delta)`*** method, when invoked, performs the following steps:

1. Let *tableaddr* be **this**.[[Table]].
2. Let *store* be the surrounding agent's associated store.
3. Let *initialSize* be table_size(*store*, *tableaddr*).
4. Let *result* be table_grow(*store*, *tableaddr*, *delta*).
5. If *result* is error, throw a `RangeError` exception.

    Note: The above exception can happen due to either insufficient memory or an invalid size parameter.

6. Set the surrounding agent's associated store to *result*.
7. Return *initialSize*.

The getter of the ***`length`*** attribute of `Table`, when invoked, performs the following steps:

1. Let *tableaddr* be **this**.[[Table]].
2. Let *store* be the surrounding agent's associated store.

2. Let *store* be the surrounding agent's associated store.

3. Return table_size(*store*, *tableaddr*).

The ***get(index)*** method, when invoked, performs the following steps:

1. Let *tableaddr* be **this**.[[Table]].
2. Let *store* be the surrounding agent's associated store.
3. Let *result* be table_read(*store*, *tableaddr*, *index*).
4. If *result* is error, throw a `RangeError` exception.
5. Let *function* be the result of creating a new Exported Function from *result*.
6. Return *function*.

The ***set(index, value)*** method, when invoked, performs the following steps:

1. Let *tableaddr* be **this**.[[Table]].
2. If *value* is null, let *funcaddr* be an empty function element.
3. Otherwise,

    1. If *value* does not have a [[FunctionAddress]] internal slot, throw a `TypeError` exception.
    2. Let *funcaddr* be *value*.[[FunctionAddress]].

4. Let *store* be the surrounding agent's associated store.
5. Let *store* be table_write(*store*, *tableaddr*, *index*, *funcaddr*).
6. If *store* is error, throw a `RangeError` exception.
7. Set the surrounding agent's associated store to *store*.

§ 4.5. Globals

```
enum ValueType {
  "i32",
  "i64",
  "f32",
  "f64"
};
```

Note: this type may be extended with additional cases in future versions of WebAssembly.

```
dictionary GlobalDescriptor {
  required ValueType value;
  boolean mutable = false;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
```

```
interface Global {
  constructor(GlobalDescriptor descriptor, optional any v);
  any valueOf();
  attribute any value;
};
```

A `Global` object represents a single global instance which can be simultaneously referenced by multiple `Instance` objects. Each `Global` object has one internal slot:

- [[Global]] : a global address

To ***initialize a global object*** *global* from a global address *globaladdr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Global object cache.
2. Assert: *map*[*globaladdr*] doesn't exist.
3. Set *global*.[[Global]] to *globaladdr*.
4. Set *map*[*globaladdr*] to *global*.

To ***create a global object*** from a global address *globaladdr*, perform the following steps:

1. Let *map* be the current agent's associated Global object cache.
2. If *map*[*globaladdr*] exists,
    1. Return *map*[*globaladdr*].
3. Let *global* be a new `Global`.
4. Initialize *global* from *globaladdr*.
5. Return *global*.

The algorithm ***ToValueType***(*s*) performs the following steps:

1. If *s* equals "i32", return i32.
2. If *s* equals "i64", return i64.
3. If *s* equals "f32", return f32.
4. If *s* equals "f64", return f64.

The algorithm ***DefaultValue***(*valuetype*) performs the following steps:

1. If *valuetype* equals i32, return i32.const 0.
2. If *valuetype* equals i64, return i64.const 0.
3. If *valuetype* equals f32, return f32.const 0.
4. If *valuetype* equals f64, return f64.const 0.
5. Assert: This step is not reached.

The ***`Global(descriptor, v)`*** constructor, when invoked, performs the following steps:

1. Let *mutable* be *descriptor*["mutable"].
2. Let *valuetype* be ToValueType(*descriptor*["value"]).
3. If *v* is undefined,
    1. Let *value* be DefaultValue(*valuetype*).
4. Otherwise,
    1. Let *value* be ToWebAssemblyValue(*v*, *valuetype*).
5. If *mutable* is true, let *globaltype* be var *valuetype*; otherwise, let *globaltype* be const *valuetype*.
6. Let *store* be the current agent's associated store.
7. Let (*store*, *globaladdr*) be global_alloc(*store*, *globaltype*, *value*).
8. Set the current agent's associated store to *store*.
9. Initialize **this** from *globaladdr*.

The algorithm ***GetGlobalValue***(`Global` *global*) performs the following steps:

1. Let *store* be the current agent's associated store.
2. Let *globaladdr* be *global*.[[Global]].
3. Let *value* be global_read(*store*, *globaladdr*).
4. Return ToJSValue(*value*).

The getter of the ***`value`*** attribute of `Global`, when invoked, performs the following steps:

1. Return GetGlobalValue(**this**).

The setter of the value attribute of `Global`, when invoked, performs the following steps:

1. Let *store* be the current agent's associated store.
2. Let *globaladdr* be **this**.[[Global]].
3. Let *mut valuetype* be global_type(*store*, *globaladdr*).
4. If *mut* is const, throw a `TypeError`.
5. Let *value* be ToWebAssemblyValue(**the given value**, *valuetype*).
6. Let *store* be global_write(*store*, *globaladdr*, *value*).
7. If *store* is error, throw a `RangeError` exception.
8. Set the current agent's associated store to *store*.

The ***`valueOf()`*** method, when invoked, performs the following steps:

1. Return GetGlobalValue(**this**).

§ 4.6. Exported Functions

A WebAssembly function is made available in JavaScript as an ***Exported Function***. Exported Functions are Built-in

A WebAssembly function is made available in JavaScript as an ***Exported Function***. Exported Functions are Built-in Function Objects which are not constructors, and which have a [[FunctionAddress]] internal slot. This slot holds a function address relative to the surrounding agent's associated store.

The ***name of the WebAssembly function*** *funcaddr* is found by performing the following steps:

1. Let *store* be the surrounding agent's associated store.
2. Let *funcinst* be *store*.funcs[*funcaddr*].
3. If *funcinst* is of the form {type *functype*, hostcode *hostfunc*},
    1. Assert: *hostfunc* is a JavaScript object and IsCallable(*hostfunc*) is true.
    2. Let *index* be the index of the host function *funcaddr*.
4. Otherwise,
    1. Let *moduleinst* be *funcinst*.module.
    2. Assert: *funcaddr* is contained in *moduleinst*.funcaddrs.
    3. Let *index* be the index of *moduleinst*.funcaddrs where *funcaddr* is found.
5. Return ! ToString(*index*).

To create ***a new Exported Function*** from a WebAssembly function address *funcaddr*, perform the following steps:

1. Let *map* be the surrounding agent's associated Exported Function cache.
2. If *map*[*funcaddr*] exists,
    1. Return *map*[*funcaddr*].
3. Let *steps* be "call the Exported Function *funcaddr* with arguments."
4. Let *realm* be the current Realm.
5. Let *function* be CreateBuiltinFunction(*realm*, *steps*, %FunctionPrototype%, « [[FunctionAddress]] »).
6. Set *function*.[[FunctionAddress]] to *funcaddr*.
7. Let *store* be the surrounding agent's associated store.
8. Let *functype* be func_type(*store*, *funcaddr*).
9. Let [*paramTypes*] → [*resultTypes*] be *functype*.
10. Let *arity* be *paramTypes*'s size.
11. Perform ! SetFunctionLength(*function*, *arity*).
12. Let *name* be the name of the WebAssembly function *funcaddr*.
13. Perform ! SetFunctionName(*function*, *name*).
14. Set *map*[*funcaddr*] to *function*.
15. Return *function*.

To ***call an Exported Function*** with function address *funcaddr* and a list of JavaScript arguments *argValues*, perform the following steps:

1. Let *store* be the surrounding agent's associated store.

2. Let *functype* be func_type(*store*, *funcaddr*).
3. Let [*parameters*] → [*results*] be *functype*.
4. Let *args* be « ».
5. Let *i* be 0.
6. For each *t* of *parameters*,
    1. If *argValues*'s size > *i*, let *arg* be *argValues*[*i*].
    2. Otherwise, let *arg* be undefined.
    3. Append ToWebAssemblyValue(*arg*, *t*) to *args*.
    4. Set *i* to *i* + 1.
7. Let (*store*, *ret*) be the result of func_invoke(*store*, *funcaddr*, *args*).
8. Set the surrounding agent's associated store to *store*.
9. If *ret* is error, throw an exception. This exception should be a WebAssembly `RuntimeError` exception, unless otherwise indicated by the WebAssembly error mapping.
10. Let *outArity* be the size of *ret*.
11. If *outArity* is 0, return undefined.
12. Otherwise, if *outArity* is 1, return ToJSValue(*ret*[0]).
13. Otherwise,
    1. Let *values* be « ».
    2. For each *r* of *ret*,
        1. Append ToJSValue(*r*) to *values*.
    3. Return CreateArrayFromList(*values*).

Note: Calling an Exported Function executes in the [[Realm]] of the callee Exported Function, as per the definition of built-in function objects.

Note: Exported Functions do not have a [[Construct]] method and thus it is not possible to call one with the `new` operator.

To ***run a host function*** from the JavaScript object *func*, type *functype*, and list of WebAssembly values *arguments*, perform the following steps:

1. Let [*parameters*] → [*results*] be *functype*.
2. Let *jsArguments* be « ».
3. For each *arg* of *arguments*,
    1. Append ! ToJSValue(*arg*) to *jsArguments*.
4. Let *ret* be ? Call(*func*, undefined, *jsArguments*).

5. Let *resultsSize* be *results*'s size.
6. If *resultsSize* is 0, return « ».
7. Otherwise, if *resultsSize* is 1, return « ? ToWebAssemblyValue(*ret*, *results*[0]) ».
8. Otherwise,

    1. Let *method* be ? GetMethod(*ret*, @@iterator).
    2. If *method* is undefined, throw a `TypeError`.
    3. Let *values* be ? IterableToList(*ret*, *method*).
    4. Let *wasmValues* be a new, empty list.
    5. If *values*'s size is not *resultsSize*, throw a `TypeError` exception.
    6. For each *value* and *resultType* in *values* and *results*, paired linearly,

        1. Append ToWebAssemblyValue(*value*, *resultType*) to *wasmValues*.

    7. Return *wasmValues*.

To ***create a host function*** from the JavaScript object *func* and type *functype*, perform the following steps:

1. Assert: IsCallable(*func*).
2. Let *stored settings* be the incumbent settings object.
3. Let *hostfunc* be a host function which performs the following steps when called with arguments *arguments*:

    1. Let *realm* be *func*'s associated Realm.
    2. Let *relevant settings* be *realm*'s settings object.
    3. Prepare to run script with *relevant settings*.
    4. Prepare to run a callback with *stored settings*.
    5. Let *result* be the result of running a host function from *func*, *functype*, and *arguments*.
    6. Clean up after running a callback with *stored settings*.
    7. Clean up after running script with *relevant settings*.
    8. Assert: *result*.[[Type]] is throw or normal.
    9. If *result*.[[Type]] is throw, then trigger a WebAssembly trap, and propagate *result*.[[Value]] to the enclosing JavaScript.
    10. Otherwise, return *result*.[[Value]].

4. Let *store* be the surrounding agent's associated store.
5. Let (*store*, *funcaddr*) be func_alloc(*store*, *functype*, *hostfunc*).
6. Set the surrounding agent's associated store to *store*.
7. Return *funcaddr*.

The algorithm ***ToJSValue***(*w*) coerces a WebAssembly value to a JavaScript value by performing the following steps:

1. If *w* is of the form i64.const *i64*,

1. Let *v* be signed_64(*i64*).
2. Return a BigInt representing the mathematical value *v*.

2. If *w* is of the form i32.const *i32*, return the Number value for signed_32(*i32*).
3. If *w* is of the form f32.const *f32*, return the Number value for *f32*.
4. If *w* is of the form f64.const *f64*, return the Number value for *f64*.

Note: Number values which are equal to NaN may have various observable NaN payloads; see NumberToRawBytes for details.

The algorithm ***ToWebAssemblyValue***(*v*, *type*) coerces a JavaScript value to a WebAssembly value by performing the following steps:

1. If *type* is i64,
   1. Let *i64* be ? ToBigInt64(*v*).
   2. Return i64.const *i64*.
2. If *type* is i32,
   1. Let *i32* be ? ToInt32(*v*).
   2. Return i32.const *i32*.
3. If *type* is f32,
   1. Let *f32* be ? ToNumber(*v*) rounded to the nearest representable value using IEEE 754-2008 round to nearest, ties to even mode.
   2. Return f32.const *f32*.
4. If *type* is f64,
   1. Let *f64* be ? ToNumber(*v*).
   2. Return f64.const *f64*.

§ 4.7. Error Objects

WebAssembly defines the following Error classes: ***`CompileError`***, ***`LinkError`***, and ***`RuntimeError`***.

When the namespace object for the `WebAssembly` namespace is created, the following steps must be run:

1. Let *namespaceObject* the namespace object.
2. For each *error* of « "CompileError", "LinkError", "RuntimeError" »,
   1. Let *constructor* be a new object, implementing the NativeError Object Structure, with *NativeError* set to *error*.
   2. ! CreateMethodProperty(*namespaceObject*, *error*, *constructor*).

Note: This defines `CompileError`, `LinkError`, and `RuntimeError` classes on the `WebAssembly` namespace, which are produced by the APIs defined in this specification. They expose the same interface as native JavaScript errors like `TypeError` and `RangeError`.

Note: It is not currently possible to define this behavior using Web IDL.

§ 5. Error Condition Mappings to JavaScript

Running WebAssembly programs encounter certain events which halt execution of the WebAssembly code. WebAssembly code (currently) has no way to catch these conditions and thus an exception will necessarily propagate to the enclosing non-WebAssembly caller (whether it is a browser, JavaScript or another runtime system) where it is handled like a normal JavaScript exception.

If WebAssembly calls JavaScript via import and the JavaScript throws an exception, the exception is propagated through the WebAssembly activation to the enclosing caller.

Because JavaScript exceptions can be handled, and JavaScript can continue to call WebAssembly exports after a trap has been handled, traps do not, in general, prevent future execution.

§ 5.1. Stack Overflow

Whenever a stack overflow occurs in WebAssembly code, the same class of exception is thrown as for a stack overflow in JavaScript. The particular exception here is implementation-defined in both cases.

Note: ECMAScript doesn't specify any sort of behavior on stack overflow; implementations have been observed to throw `RangeError`, InternalError or Error. Any is valid here.

§ 5.2. Out of Memory

Whenever validation, compilation or instantiation run out of memory, the same class of exception is thrown as for out of memory conditions in JavaScript. The particular exception here is implementation-defined in both cases.

Note: ECMAScript doesn't specify any sort of behavior on out-of-memory conditions; implementations have been observed to throw OOMError and to crash. Either is valid here.

ISSUE 1 A failed allocation of a large table or memory may either result in

- a `RangeError`, as specified in the `Memory` `grow()` and `Table` `grow()` operations
- returning -1 as the memory.grow instruction
- UA-specific OOM behavior as described in this section.

In a future revision, we may reconsider more reliable and recoverable errors for allocations of large amounts of

memory.

See Issue 879 for further discussion.

§ 6. Implementation-defined Limits

The WebAssembly core specification allows an implementation to define limits on the syntactic structure of the module. While each embedding of WebAssembly may choose to define its own limits, for predictability the standard WebAssembly JavaScript Interface described in this document defines the following exact limits. An implementation must reject a module that exceeds one of the following limits with a `CompileError`: In practice, an implementation may run out of resources for valid modules below these limits.

- The maximum size of a module is 1073741824 bytes (1 GiB).
- The maximum number of types defined in the types section is 1000000.
- The maximum number of functions defined in a module is 1000000.
- The maximum number of imports declared in a module is 100000.
- The maximum number of exports declared in a module is 100000.
- The maximum number of globals defined in a module is 1000000.
- The maximum number of data segments defined in a module is 100000.
- The maximum number of tables, including declared or imported tables, is 1.
- The maximum number of table entries in any table initialization is 10000000.
- The maximum number of memories, including declared or imported memories, is 1.
- The maximum number of parameters to any function or block is 1000.
- The maximum number of return values for any function or block is 1000.
- The maximum size of a function body, including locals declarations, is 7654321 bytes.
- The maximum number of locals declared in a function, including implicitly declared as parameters, is 50000.

An implementation must throw a `RuntimeError` if one of the following limits is exceeded during runtime: In practice, an implementation may run out of resources for valid modules below these limits.

- The maximum size of a table is 10000000.
- The maximum number of pages of a memory is 65536.

§ 7. Security and Privacy Considerations

This section is non-normative.

This document defines a host environment for WebAssembly. It enables a WebAssembly instance to import JavaScript objects and functions from an import object, but otherwise provides no access to the embedding environment. Thus a

WebAssembly instance is bounds to the same constraints as JavaScript.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

> EXAMPLE 1
>
> This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

> Note, this is an informative note.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[ECMASCRIPT] defines the following terms:

- %functionprototype%
- @@iterator
- ArrayBuffer
- CreateMethodProperty
- RangeError
- TypeError
- agent
- agent cluster
- bigint
- built-in function objects
- call
- createarrayfromlist
- createbuiltinfunction
- createdataproperty
- current realm
- data block
- detacharraybuffer
- get
- getmethod
- iscallable
- iterabletolist
- the number value
- tobigint64
- toint32
- tonumber
- tostring
- type

[ENCODING] defines the following terms:

- utf-8 decode without bom or fail

[HTML] defines the following terms:

- clean up after running a callback
- clean up after running script
- in parallel
- incumbent settings object
- prepare to run a callback
- prepare to run script
- queue a task
- settings object
- task source

[INFRA] defines the following terms:

- append
- exist
- .

iterabletolist
nativeerror object structure
numbertorawbytes
objectcreate
setfunctionlength
setfunctionname
setintegritylevel
surrounding agent
is empty
iterate
list
ordered map
set
size

[WEBASSEMBLY] defines the following terms:

address
const
current frame
custom section
customsec
error
external value
f32
f32.const
f64
f64.const
func
func_alloc
func_invoke
func_type
function address
function element
global
global address
global instance
global_alloc
global_read
global_type
global_write
host function
i32
i32.const
i64
i64.const
import
imports
instance_export
mem
mem_alloc
module_decode
module_exports
module_imports
module_instantiate
module_validate
sequence
signed_32
signed_64
store
store_init
table
table address
table instance
table type
table_alloc
table_grow
table_read
table_size
table_write
valid
var
webassembly module validation
webassembly value

mem_grow

mem_size

memaddrs

memory address

memory instance

memory.grow

module (for frame)

module grammar

[WebIDL] defines the following terms:

BufferSource

DOMString

EnforceRange

Exposed

Function

LegacyNamespace

USVString

a new promise

associated realm

boolean

create a namespace object

get a copy of the buffer source

implements

namespace object

new

object

reject

resolve

throw

unsigned long

upon fulfillment

upon rejection

§ References

§ Normative References

[ECMASCRIPT]
ECMAScript Language Specification. URL: https://tc39.es/ecma262/

[ENCODING]
Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[RFC2119]

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[WEBASSEMBLY]

WebAssembly Core Specification. Draft. URL: https://webassembly.github.io/spec/core/

[WebIDL]

Boris Zbarsky. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

§ IDL Index

```
dictionary WebAssemblyInstantiatedSource {
    required Module module;
    required Instance instance;
};

[Exposed=(Window,Worker,Worklet)]
namespace WebAssembly {
    boolean validate(BufferSource bytes);
    Promise<Module> compile(BufferSource bytes);

    Promise<WebAssemblyInstantiatedSource> instantiate(
        BufferSource bytes, optional object importObject);

    Promise<Instance> instantiate(
        Module moduleObject, optional object importObject);
};

enum ImportExportKind {
  "function",
  "table",
  "memory",
  "global"
};

dictionary ModuleExportDescriptor {
  required USVString name;
  required ImportExportKind kind;
  // Note: Other fields such as signature may be added in the future.
};

dictionary ModuleImportDescriptor {
  required USVString module;
  required USVString name;
  required ImportExportKind kind;
};
```

```
[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Module {
  constructor(BufferSource bytes);
  static sequence<ModuleExportDescriptor> exports(Module moduleObject);
  static sequence<ModuleImportDescriptor> imports(Module moduleObject);
  static sequence<ArrayBuffer> customSections(Module moduleObject, DOMString sectionName
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Instance {
  constructor(Module module, optional object importObject);
  readonly attribute object exports;
};

dictionary MemoryDescriptor {
  required [EnforceRange] unsigned long initial;
  [EnforceRange] unsigned long maximum;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Memory {
  constructor(MemoryDescriptor descriptor);
  unsigned long grow([EnforceRange] unsigned long delta);
  readonly attribute ArrayBuffer buffer;
};

enum TableKind {
  "anyfunc",
  // Note: More values may be added in future iterations,
  // e.g., typed function references, typed GC references
};

dictionary TableDescriptor {
  required TableKind element;
  required [EnforceRange] unsigned long initial;
  [EnforceRange] unsigned long maximum;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Table {
  constructor(TableDescriptor descriptor);
  unsigned long grow([EnforceRange] unsigned long delta);
  Function? get([EnforceRange] unsigned long index);
  void set([EnforceRange] unsigned long index, Function? value);
  readonly attribute unsigned long length;
};

enum ValueType {
  "i32",
  "i64",
  "f32",
  "f64"
};
```

```
};

dictionary GlobalDescriptor {
  required ValueType value;
  boolean mutable = false;
};

[LegacyNamespace=WebAssembly, Exposed=(Window,Worker,Worklet)]
interface Global {
  constructor(GlobalDescriptor descriptor, optional any v);
  any valueOf();
  attribute any value;
};
```

§ Issues Index

ISSUE 1 A failed allocation of a large table or memory may either result in

- a `RangeError`, as specified in the `Memory.grow()` and `Table.grow()` operations
- returning -1 as the memory.grow instruction
- UA-specific OOM behavior as described in this section.

In a future revision, we may reconsider more reliable and recoverable errors for allocations of large amounts of memory.

See Issue 879 for further discussion.

↵