CommonJS

Draft of Standards-Track Draft Proposal for Specification

Endorsed by Nobody

Last Update
January 13, 2011

Editors
Wes Garland,
wes@page.ca

Florian Traverse,
florian.traverse@atosorigin.com

Another Name Here,
local-part@domain

Another Name Here,
local-part@domain

Modules/2.0

Universal Module Authoring Format

Securable Modules

Dependencies

Extensibility

Portability

CommonJS

1. Introduction

This CommonJS specification draws together the highly successful and widely-deployed Modules/1.0 through Modules/1.1.1 specifications, three years of implementation expertise on a variety of host environments, and ideas from the CommonJS Mailing List and IRC Channel. This specification is a super-set of Modules/Wrappers, and also draws ideas from the following proposals:

- CommonJS Asynchronous Module Definition (implemented in RequireJS, Dojo, Nodules, and Transporter)
- CommonJS Async/A (implemented in Yabble)
- CommonJS Transports/D (implemented in RequireJS and Yabble)
- CommonJS Loaders/B
- ECMAScript.org strawman:modules_primordials
- ECMAScript.org strawman:simple_modules

This document attempts to guide runtime implementations toward CommonJS environment interoperability, and provides guidelines for users wanting to write portable CommonJS modules and programs.

> **Note**
>
> Section 1. of this document is non-normative.

1.1 Scope

This document describes the CommonJS environment and module authoring format. Environment bootstrapping and main module loading are not specified in this document, however recommendations are made to guide implementers.

1.2 Conformance

An execution environment conforming to this specification must provide the CommonJS environment as described, execute modules with the described semantics, and implement all APIs in this specification. Some APIs in this specification have formal methods for indicating that they are not supported in a particular environment. Indicating lack of support for these APIs is sufficient to achieve conformance.

A conformant environment may include extra APIs, short hand notation, alternate module loading mechanics and various optimizations. This specification does not limit the functionality of conformant environments, except to insure that all APIs described are implemented.

Modules which claim conformance with this specification must not make use of features or interfaces which are not described in ECMAScript-262 Edition 3 or this specification without first feature-testing their availability, or taking alternative steps to insure said features are available to the host environment.

Conformant environments are not limited to running only conformant modules.

1.3 Goals

Primary Goals

This specification attempts to define a non-restrictive CommonJS environment suitable for use by web pages, web browsers, GUI toolkits, application servers, batch systems programming, application-embedded ECMAScript, and other host environments by addressing the following issues:

- Creation of a single module authoring format that can be supported on all conformant ECMAScript environments, having the following characteristics
- Able to maintain Securable Modules properties
- Interoperable with Modules/1.1 in the same CommonJS Environment
- Trivial to convert Modules/1.1 modules to Modules/2.0 format
- Specification of dependencies without the need for static analysis, including support for computed dependencies, without restricting the host environment's concurrency model
- More rigorous definition of the CommonJS Environment
- To clarify the availability and role of main modules ("program modules")
- To enable the 'return exports' idiom
- To clarify the scope of the '`require`' symbol, such that it may be used to share a module's authority
- To adjust the scope chain definition from module code downward (toward the global object) in a way which is compatible with other module standards under consideration by ECMA TC-39.

Secondary Goals

Note: Both "Primary" and "Secondary" goals will be merged into a "Goals" section as this document makes it's way through the draft / ratification process.

- To enable end-user or plug-in patching of the CommonJS Environment in a uniform manner across all platforms which support this functionality, and to provide a means to detect whether or not a given CommonJS environment supports it
- To enable the replacement of the exports object with an alternate object, including Function objects.
- More rigorous definition of the CommonJS Environment
- To make uniform recommendations for program bootstrapping
- To standardize an API for explicit lazy-loading of modules

1.4 Modules/2.0 Name

This specification intends to signal to module developers which standard they should adhere to when developing modules for the widest possible distribution. Calling this specification "2.0" identifies it in a way which will encourage existing implementations to add this functionality.

While this proposal is effectively "Securable Modules" at it's core, it is not a trivial variant. The group of CommonJS Environments implementing this specification will have at their intersection a more functional core.

Version numbers are also used to signal compatibility; modules written in this idiom are not compatible with platforms supporting Modules/1.1. Additionally, this specification does not require that platforms supporting this specification are able to execute Modules/1.1 modules. As such, a source-compatibility barrier exists that warrants a major version increase.

1.5 Definitions

Note

This table is non-normative.

CommonJS environment	The execution environment described in §2 of this specification
Host environment	The underlying ECMAScript environment, upon which the CommonJS environment is built
Module	Stand-alone code unit
To provide a module	Make a module available for use such that referencing its exports or initializing it is possible without blocking
To load a module	To fetch the source code of a module from a resource
Resource	File, database entry, etc, which may contain module source code
module identifier	A non-unique string which, in conjunction with its calling environment, identifies a unique module
`module.id`	A unique string which canonically identifies a particular module

2. The CommonJS Environment

The CommonJS Environment consists of an ECMAScript execution environment embedded in an application such as a web browser or scripting host, known as the host environment.

In an addition to the ECMAScript interpreter, the CommonJS environment provides a facility for loading and executing ECMAScript known as the CommonJS module system, and bootstrapping facilities that allow us to initialize and execute CommonJS scripts.

2.1 The CommonJS Module System

The CommonJS module system performs three main tasks:

2.1.1 Provide modules to the environment and memoize their exports

2.1.2 Resolve dependencies between modules

2.1.3 Allow scripts to access properties exported from modules which have been provided to the environment

2.2 Characteristics of all modules

All modules in the in the CommonJS Environment:

2.2.1 Have a unique require function object.

2.2.2 Have a unique exports object.

2.2.3 Have a unique module object with a unique `module.id`.

2.2.4 Behave as singletons. This means that two calls to require() which reference the same module's exports must return the same object.

2.2.5 Do not affect the scope chain of other modules.

2.3 Characteristics of the Main Module

The main module, sometimes called the "program module" in server-side environments, consists of the module initially executed by the CommonJS Environment.

The main module has the same characteristics as any other module, however since it is loaded via environment-specific facilities (rather than the require() function), it is not always possible to know the name of the resource providing the main module. Environments that cannot determine the name of the main module may set its `module.id` to the empty string.

Examples of environment-specific modules include code found inside `<SCRIPT>` tags embedded directly in an HTML document, or code below the she-bang (`#!`) line of a script launched via the UNIX® `exec()` system call. Conforming environments may also provide a way to execute arbitrary resources as the main module.

2.3.1 The main module has the same characteristics as other modules, except

2.3.1.1 The main module's factory function is invoked before any other module, through environment-specific means

2.3.2 If the main module cannot be referenced via other APIs in this specification, it may use the empty string as its `module.id`.

Only zero or one main modules per instance of the CommonJS Environment are permitted.

2.4 Concurrency Model

Conformant CommonJS environments may be implemented in a variety of concurrency models. This specification does not mandate any particular concurrency model, however it does differentiate in places between event-loop systems and other systems. This differentiation is made such that module authors do not need to be aware of the underlying concurrency model when writing conformant modules.

2.5 ECMAScript Language Conformance

The host environment must conform to ECMA-262 Edition 3 or ECMA-262 Edition 5. Host environments conforming to ECMA-262 Edition 3 must supply shims to emulate Edition 5 wherever possible.

Modules conforming to this specification must be written to target the subset of Edition 5 that can be implemented by shimming Edition 3.

The mandatory Edition 5 interfaces are:

Interface	**ECMA 262 Edition 5 §**
`Array.isArray`	15.4.3.2
`Array.prototype.indexOf`	15.4.4.14
`Array.prototype.lastIndexOf`	15.4.4.15
`Array.prototype.every`	15.4.4.16
`Array.prototype.some`	15.4.4.17
`Array.prototype.forEach`	15.4.4.18
`Array.prototype.map`	15.4.4.19
`Array.prototype.filter`	15.4.4.20
`Array.prototype.reduce`	15.4.4.21
`Array.prototype.reduceRight`	15.4.4.22
`Object.getPrototypeOf`	15.2.3.2
`Object.getOwnPropertyDescriptor`	15.2.3.3
`Object.getOwnPropertyNames`	15.2.3.4
`Object.create`	15.2.3.5
`Object.defineProperty`	15.2.3.6

`Object.defineProperties`	15.2.3.7
`Object.seal`	15.2.3.8
`Object.freeze`	15.2.3.9
`Object.preventExtensions`	15.2.3.10
`Object.isSealed`	15.2.3.11
`Object.isFrozen`	15.2.3.12
`Object.isExtensible`	15.2.3.13
`Object.keys`	15.2.3.14
`Date.now`	15.9.4.4
`Date.prototype.toISOString`	15.9.5.43
`Date.prototype.toJSON`	15.9.5.44
`Function.prototype.bind`	15.3.4.5
`String.prototype.trim`	15.5.4.20
`JSON`	15.12

3. CommonJS Modules

A CommonJS module code section is the group of expressions contained within a module factory function. Any series of ECMAScript expressions that can be parsed as an ECMAScript `FunctionBody` may be used as a module code section.

CommonJS module code sections defined in this specification are virtually identical to modules in the CommonJS Modules/1.1.1 specification; that specification is largely embedded into §3.2 and §3.3 of this document.

CommonJS modules may be stored as any type of resource, including local files, remote URLs, database entries, and so on. The storage mechanism of a module is relevant only to the environment's built-in module provider and not to any consumer.

3.1 Module Declaration

3.1.1 Modules are declared with a call to `module.declare()`, which accepts an optional dependency array and a module factory function

3.1.1.1 If the first argument is a function, the module is treated as though it has inferred dependencies.

3.1.1.2 If the first argument is not a function, then it is treated as the dependency array.

3.1.1.3 Conformant environments must process the dependency array when it is present.

There is no requirement to process inferred dependencies, however the CommonJS environment may infer dependencies through any means the platform authors feel is appropriate, such as performing static analysis on the module code body to identify calls to `require()`.

3.1.2 Modules are declared with a call to `module.declare`, which accepts as one of its parameters a module factory function.

3.1.3 Module factory functions contain module code sections.

3.1.4 Module code sections which are also valid CommonJS Modules/1.1.1 modules must behave as though the contained module were executed in a CommonJS Modules/1.1.1 environment.

3.1.5 Resources (e.g. files) which provide modules must do so with a single ECMAScript `SourceElement` (see: ECMA-262 section 14)

3.1.6 This `SourceElement` is written in source format without a trailing semi-colon.

3.1.7 The module provider may throw an exception when multi-`SourceElement` resources or trailing semi-colons are encountered.

3.1.8 The module factory function accepts the arguments require, exports, and module, returning either nothing (undefined) or an alternate exports object.

3.1.9 The module provider satisfies all dependencies before the module is provided to the environment.

3.1.10 The `exports` object is memoized by the module provider (made available to `require`) before the module factory function is invoked.

Notes

The single-`SourceElement` rules written to allow validation of module resources by surrounding the module declaration with parentheses and invoking `eval()`, and to ease composition of resources for transporting multiple modules inside an ECMAScript `Object` or `Array` literal.

The `exports` object memoization step in 2.4.9 is necessary for the correct resolution of transitive and circular dependencies, and is consistent with Modules/1.0.

3.2 Module Code Section

This section addresses how module code sections should be written in order to be interoperable among a class of module systems that can be both client and server side, secure or insecure, implemented today or supported by future systems with syntax extensions.

These modules are offered privacy of their top scope, facility for importing singleton objects from other modules, and exporting their own API. By implication, this specification defines the minimum features that a module system must provide in order to support interoperable modules.

3.2.1 In a module code section, there is a free variable `require`, that is a `Function` object.

3.2.1.1 The `require` function accepts a module identifier.

3.2.1.2 `require` returns the exported API of the foreign module.

3.2.1.3 If there is a dependency cycle, the foreign module may not have finished executing at the time it is required by one of its transitive dependencies; in this case, the object returned by `require` must contain at least the exports that the foreign module has prepared before the call to require that led to the current module's execution.

3.2.1.4 If the requested module cannot be returned, `require` must throw an error.

3.2.2 The `require` function may have a `main` property which represents the top-level `module` object of the program. This property must be referentially identical to the `module` object of the main program.

Deprecation

This property is deprecated, but included for backwards compatibility with Modules/1.1.1.

3.2.2.1 The `require` function may have a `paths` attribute, defined in §5.2

3.2.2.2 The `paths` property must not exist in `sandbox` (a secured module system).

In a module code section, there is a free variable called `exports`, that is an object that the module may add its API to as it executes.

3.2.2.3 Modules must use the `exports` object as the only means of exporting, except during module initialization

3.2.2.4 During module initialization, modules may return an alternate exports value, which may be an object or a function object

3.2.2.5 Conformant modules which return alternate exports may not invoke the require function during module initialization

3.2.3 In a module code section, there must be a free variable `module`, that is an `Object`.

3.2.3.1 The `module` object must have a `id` property that is the top-level "`id`" of the module. The `id` property must be such that r`equire(module.id)` will return the exports object from which the `module.id` originated. (That is to say `module.id` can be passed to another module, and requiring that must return the original module). When feasible this property should be read-only, don't delete.

3.2.3.2 The `module` object may have a `uri` String that is the fully-qualified URI to the resource from which the module was created. The `uri` property must not exist in a sandbox.

Deprecation

This property is deprecated, but included for backwards compatibility with Modules/1.1.1.

3.3 Module Identifiers

3.3.1 A module identifier is a String of "terms" delimited by forward slashes.

3.3.2 A term must be a series of one or more module identifier characters, "`.`", or "`..`".

3.3.2.1 module identifier characters are the set of lowercase characters, numbers, the underscore character, the dash character, and the period.

3.3.3 Module identifiers must not have file-name extensions like "`.js`".

3.3.4 Module identifiers may be "relative" or "top-level". A module identifier is "relative" if the first term is "`.`" or "`..`".

3.3.5 Top-level identifiers are resolved off the conceptual module name space root.

3.3.6 Relative identifiers are resolved relative to the identifier of the module for which "`require`" was instantiated.

3.4 Module Factory Functions

3.4.1 Module factory functions contain the module code sections.

Each module factory function accepts as its first three arguments `require`, `exports`, and `module`.

3.4.1.1 Conformant modules will spell the arguments out in full – `require`, `exports`, `module`.

3.5 Dependency Arrays

3.5.1 Module dependencies are formally described with a dependency array.

3.5.2 Dependency arrays contain only `String` or `Object` values.

3.5.3 It is not required that the dependency array in a module declaration be an array literal.

3.5.4 Each array element which is a `String` value represents a dependency upon a module whose exports may be referenced by passing the array element as an argument to `require()` from within the module code section, ignoring masking conflicts with labeled dependencies.

3.5.5 Each array element which is an object contains one or more labeled dependencies

3.5.5.1 Each labeled dependency object has `ownProperties` whose values are treated the same way as strings values in the dependency array.

3.5.5.2 Each label (`ownProperty` name) represents a name which can be used by require() to reference the dependent module's exports from within the module code body.

3.5.5.3 Labeled dependency names that conflict with other module identifiers have precedence.

3.5.5.4 Labeled dependencies must not affect module name resolution in other modules.

3.5.6 There is no limit on the number of dependencies specified in the dependencies array, nor the number of dependencies specified in a labeled dependency object.

3.5.7 A reference to the dependency array is made available to the module as `module.dependencies`. If the module was declared without a dependency array, this value is undefined.

3.6 Module Providers

The term "module provider" describes a facility which loads modules and module dependencies into the CommonJS environment so that their exports can be referenced by a call to `require()`.

Module provider plug-ins are possible with this specification, allowing third parties to create module providers that extend the base functionality of the CommonJS environment. These plug-ins, which are defined by overriding properties on `module.constructor.prototype`, use calls in the require namespace to notify the CommonJS environment of newly-provided modules. Executing the module factory function is left to the CommonJS environment.

Notes

A conforming environment may use a just-in-time-provide optimization which delays loading and providing of modules until their exports are referenced by require.
A conforming environment may also use the all-module-provide optimization, which behaves as though the main module depends on all possible modules.
Most current Modules/1.1 CommonJS environments implement both these optimizations, as they were implied in Modules/1.0.

3.7 Extra-Module Environment

Conformant CommonJS platforms may provide a mechanism to execute ECMAScript code outside of any module's context.

This extra-module environment, which typically (but not necessarily) uses the global object as its lexical scope,

3.7.1 Has a free variable named `module`, which is an object that has the same constructor as the `module` object visible from any CommonJS module, except that

3.7.1.1 This module object has an undefined id property

3.7.2 Has a free variable named `require` which behaves exactly the same as the `require` function made available to any module, except that

3.7.2.1 relative module identifiers are undefined

3.7.2.2 The main module in an HTML page is typically, but not necessarily, declared by `module.declare`, executed from the extra-module environment. See §6 Bootstrapping , for further details.

4. Module Namespace

The `module` object, available in every module in every CommonJS environment, acts as a namespace for module- and module-system information which do not impact the securability of the module system.

This namespace, in conjunction with the `require` namespace, exposes sufficient detail about the underlying module system implementation to allow CommonJS users to modify the underlying behavior of the module system. These modifications may then implement alternative module formats, storage mechanisms, package systems, etc, on top of this specification, by overriding either properties of the local module object, or `module.constructor.prototype`.

4.1 module.declare *([], f)*

This function wraps the module's dependencies and factory function into a unit, accepting as parameters an optional array of dependencies and the module factory function.

Environments that support module provider plug-ins must resolve dependencies via `module.provide` when `module.provide` is not set to the environment's default value (i.e. it has been overridden by a plug-in).

See §3.1 for further details.

4.2 module.provide *([], f)*

This function accepts a dependency list and a callback function. It satisfies the dependencies by providing them to the environment (eventually invoking `module.declare`), and then invokes an optional callback. The dependency list format is described in §3.5.

This function

4.2.1 Iterates over the list of dependencies in no particular order

4.2.2 Ignores dependencies which have already been provided to the environment

4.2.3 Loads each dependent module and invokes it (executes `module.declare`)

4.2.3.1 This step must be implemented with `module.load` if the load property of this module object differs from the environment's default value, and the environment supports module provider plug-ins.

4.2.3.2 There is no restriction with respect to parallel loading, or interleaving of operations.

4.2.4 Determines the dependencies of all dependent modules, recursively, and provides those as well.

4.2.5 Once the dependencies have all been provided to the environment, the callback function is invoked.

4.2.6 Alternate implementations of `module.provide` (i.e. those which are part of a plug-In) must invoke the callback function with `module.eventually`.

4.3 module.id

4.3.1 Uniquely identifies a module.

4.3.2 `require(module.id)` must return this module's exports object.

4.3.3 Is referentially identical to `require.id(module identifier)`.

4.4 module.main

4.4.1 Is a reference to the main module's exports object.

4.5 module.dependencies

4.5.1 `module.dependencies` is a reference to the module's dependency declaration array.

4.5.2 This array is informative only: modification of this array has no effect on the module once the module has been provided to the environment.

4.6 module.load *(s, f)*

4.6.1 This function exists to provide a hook into the module loading process for a module provider plug-in. It is typically invoked by `module.provide`.

4.6.2 When defined, this function

4.6.2.1 Accepts a module identifier and a callback function.

4.6.2.2 Fetches the source code to the named module.

4.6.2.3 Causes the module's source code (`module.declare` statement) to be evaluated.

4.6.2.4 Invokes the callback.

CommonJS environments that do not support module provider plug-ins must set this property to false. Module provider plug-ins should test this property during initialization to determine if the environment supports plug-ins.

4.7 module.eventually *(f)*

This function exists to bridge the gap between CommonJS environments that are built on event loops, and those that are not, for the purposes of writing module provider plug-ins. This function accepts a callback function, and causes it to be invoked - eventually.

CommonJS environments built on event loops may invoke the callback function directly, or place the callback in the underlying environment's event loop.

CommonJS environments which are not built on event loops must place the callback on a pending event list.

If the CommonJS program terminates due to an uncaught exception, no callbacks are invoked. When the CommonJS program terminates normally:

4.7.1 All callbacks are invoked

4.7.2 Callbacks are invoked in first-in, first-out order

4.7.3 Callbacks are invoked using the main module's global object

4.7.4 Callbacks may use `module.eventually` to add callbacks to the end of the list

4.7.5 The behavior of the CommonJS environment is not defined if invoking a callback yields an uncaught exception

4.8 module.constructor *()*

4.8.1 Is provided automatically by the host environment, via inheritance from `Object`.

4.8.2 Is the constructor used to create all module objects in a given CommonJS environment.

4.8.3 Is not `Object`.

4.8.4 `module.constructor` and any of its properties, including `module.constructor.prototype`, may be frozen in a secure environment.

4.8.5 Overriding properties of `module.constructor.prototype` is the correct way to implement a module provider.

4.9 module.uri

This property is deprecated and the name reserved for CommonJS Modules/2.0 environments wishing to provide backwards compatibility with Modules/1.1.1 modules using this feature.

5. Require Namespace

The require symbol acts as both a function and a namespace object for providing authoritative information about the CommonJS environment. A unique instance of require is provided for each module. Modules may pass require objects to other modules in order to hand off their authority.

All knowledge about the CommonJS environment and the underlying host environment, as it pertains to the module system, is encapsulated in this namespace.

Secure environments must implement require as a frozen function with read-only, don't-delete properties.

5.1 require *(s)*

When invoked as a function, `require()`

5.1.1 Accepts a module identifier and returns a module's exports.

5.1.2 If a module has been provided to the environment, but the module's factory function has not been executed, the factory function is executed before the call returns

5.1.3 If a module has not been provided to the environment, `require()` must throw an exception

5.1.3.1 It is acceptable, but not required, for the CommonJS environment to make an attempt to provide the module during the require call. This is the just-in-time provide optimization discussed in §3.5.

5.1.4 If a passed module identifier refers to a labeled dependency (see §3.5), this reference takes precedence over the method of locating module exports described in §3.2.

5.2 require.paths

5.2.1 Environments that do not support searchable module paths must not set this property.

5.2.2 Environments which support searchable modules paths must set this property to an object which is an `Array`.

5.2.3 The `require.paths` array contains a list of canonical pathnames which are searched, in order, when resolving a non-relative module identifier

5.2.4 The environment may search additional paths, before or after, the paths specified in `require.paths`

5.2.5 Replacing `require.paths` has undefined behavior; modules should use `Array` methods such as `push`, `pop`, `shift`, or `unshift` paths to adjust the array.

5.2.6 `require.paths` is referentially identical for all modules in the environment

5.3 require.id *(s)*

5.3.1 This function accepts as its argument a module identifier, returning the corresponding canonical `module.id`

5.3.2 `require(require.id(name))` is equivalent to `require(name)`

5.3.3 Can be used to generate canonical `module.id` values for modules which have not yet been provided to the environment

5.4 require.uri *(s)*

This interface provides information to module provider plug-ins wishing to override portions of the default module provider, for example, by replacing `module.load`.

5.4.1 This function accepts as its argument a module identifier or module identifier path, returning a URI as a `String` which canonically identifies the resource.

5.4.2 When the passed argument contains path separators, the final portion of the argument remains the same.

5.4.3 There is no requirement that any module *S* retrieved from the location given by `require.uri(S)` have `module.uri === require.uri(S)`.

5.4.4 There is no restriction on the protocol indicated by the URI; local inventions are permitted.

5.5 require.memoize *(s, [], f)*

This interface allows module provider plug-ins a way to provide modules to the environment.

5.5.1 This function accepts as its arguments a canonical `module.id`, dependency array, and module factory function.

5.5.2 If the module has already been provided, an exception is thrown.

5.5.3 Once this call completes, the module has been provided; i.e. a subsequent call to `require(s)` will not throw an exception.

5.6 require.isMemoized *(s)*

This interface allows module provider plug-ins a way to determine if a module has already been provided to the environment, e.g. to eliminate duplicate server requests when resolving dependencies. This function :

5.6.1 Accepts as its arguments a canonical `module.id`.

5.6.2 Returns `true` if the module has already been provided; `false` otherwise.

5.7 require.main

Deprecation

This property is deprecated and the name reserved for environments wishing to provide backwards compatibility with Modules/1.1.1 modules using this feature.

5.7.1 See §3.2.

6. Bootstrapping

Bootstrapping refers to process of initializing the CommonJS environment, loading the main module, and invoking the main module's factory function.

As bootstrapping is necessarily non-uniform across environments, this section of the specification exists only in the form of recommendations. Conformant CommonJS environments are not required to implement and of the recommendations in this section.

The section recognizes the existence of a CommonJS environment argument vector, but does not specify how it accessed.

6.1 Generic Scripting Host Environments

This environment is the simplest type of "server-side" CommonJS installation. Module resources are files on the local filesystem, and the CommonJS environment is invoked like any other program in the host environment.

It is recommended that a scripting host passed a single argument treat that argument as the filename of a program module.

It is recommended that arguments after a `"--"` argument be treated as the CommonJS environment's argument vector.

It is recommended that relative require calls from the main module are loaded relative to the directory given by the main module's filename.

It is recommended that host-environment files storing CommonJS modules end with ".js".

6.2 UNIX® File Interpreters ("she-bang")

This environment is very similar to the scripting host environment, except that rather than using the host operating system to run the scripting host, the user runs CommonJS program directly.

To loosely recap the UNIX model, the file containing the script has execute permissions, telling the operating system that it should load and run it via the `exec()` system call. When `exec()` loads the file, it discovers that the script starts with the characters "`#!`", and so parses the rest of the first line to determine the name of the file interpreter. This file interpreter is then invoked by the operating system, which is passed the name of the script to run, along with the script's argument vector.

It is recommended that during the build or install phases of the CommonJS environment, that environments supporting a UNIX file interpreter offer to create a symbolic link named "commonjs" on the user's path which points to the file interpreter. This will allow users interested in creating portable executable programs written in CommonJS to begin their scripts with the "`#! /usr/bin/env commonjs`" prologue.

It is recommended that script line numbers reported by the CommonJS environment conform to actual line numbers in the file. This means that the line beginning with "#!" is line one and the ECMAScript program begins on line two. An easy way to achieve this behavior is to instruct the host ECMAScript environment to begin parsing the script when the script's file pointer is advanced such that the next character read from the file is the first new-line character in the file.

It is recommended that the argument vector passed by the `exec()` system call to the file interpreter is treated as the CommonJS environment's argument vector.

It is recommended that relative require calls from the main module are loaded relative to the directory given by the main module's filename.

It is recommended that host-environment files storing CommonJS modules end with "`.js`".

6.3 HTML or XHTML Document

This environment is similar to the UNIX file interpreter environment in that a non-ECMAScript preamble is necessarily part of the same resource as the main module.

It is recommended that host-environment URIs storing CommonJS modules end with "`.js`".

It is recommended that the main module has the empty-string `module.id`, and is declared inline in a `<SCRIPT>` tag.

It is recommended that the environment encourage users to place the `<SCRIPT>` tag which loads the CommonJS environment as close to the top of the `<HEAD>` section in the document as possible, so that the CommonJS extra-module environment is available as soon as possible.

It is recommended that the environment implement a variation of the following algorithm (courtesy Kris Zyp) to determine the module id when modules are loaded via `<SCRIPT>` tags by examining the `SRC` attribute at runtime :

- In non-IE browsers, the `onload` event is sufficient, it always fires immediately after the script is executed.
- In IE, if the script is in the cache, it actually executes *during* the DOM insertion of the script tag, so you can keep track of which script is being requested in case `module.declare()` is called during the DOM insertion.
- In IE, if the script is not in the cache, when `module.declare()` is called you can iterate through the script tags and the currently executing one will have a `script.readyState == "interactive"`.

Note

As of this writing, "IE" refers to Microsoft Internet Explorer versions 6 through 8. It is anticipated that Internet Explorer 9 will behave the same as the other browsers on the market.

7. Sample Modules

7.1 Generic Modules

Sample Module: math.js

```
module.declare(function(require, exports, module) {
  exports.add = function() {
    var sum = 0, i = 0, args = arguments, l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum;
  }
})
```

Sample Module: increment.js

```
module.declare(['math'], function(require, exports, module) {
  var add = require('math').add;
  exports.increment = function(val) {
    return add(val, 1);
  };
})
```

7.2 Main Modules

Sample Main Module, generic scripting hosting environment: program.js

```
module.declare(["increment"], function(require, exports, module) {
  var inc = require('increment').increment;
  var a = 1;
  inc(a); // 2

  // module.id === "program"
})
```

Sample Main Module, UNIX file interpreter idiom: program.js

```
#! /usr/bin/env commonjs
module.declare(["increment"], function(require, exports, module) {
  var inc = require('increment').increment;
  var a = 1;
  inc(a); // 2

  // module.id === "program"
})
```

Sample Main Module, HTML document environment

```
<HTML>
 <HEAD>
  <SCRIPT type="text/javascript">
   module.declare(["increment"], function(require, exports, module) {
     var inc = require('increment').increment;
     var a = 1;
     inc(a); // 2
     // module.id === ''
   })
  </SCRIPT>
 </HEAD>
 <BODY onload="module.provide(['increment'], program)">
 </BODY>
</HTML>
```

8. Special Thanks

This specification would not be possible with the work of dozens of people; enumerating everyone's contribution would be a non-trivial task. We'd like to take this opportunity to thank a few key people, and hope we haven't left anyone out:

- Kevin Dangoor for getting the ServerJS/CommonJS ball rolling
- Ihab Awad, Kris Kowal for Securable Modules
- Kris Zyp, James Brantly for research in browser-side module systems
- Dean Landolt for helping to discover the `module.constructor.prototype` monkey-patch pattern
- Dean Landolt and Stefaan Coussement for inventing and refining labeled dependencies
- Brendan Eich for JavaScript
- ECMA and TC-39 for ECMAScript-1 through ECMAScript-5, and ongoing work
- All the members of the CommonJS mailing list