Ethereum Improvement Proposals

All Core Networking Interface ERC Meta Informational

This EIP has been withdrawn.

EIP-1123: Revised Ethereum Smart Contract Packaging Standard ‹›

Author	g. nicholas d'andrea, Piper Merriam, Nick Gheorghita, Danny Ryan
Discussions-To	https://github.com/ethereum/EIPs/issues/1123
Status	Withdrawn
Type	Standards Track
Category	ERC
Created	2018-06-01

Table of Contents

This ERC has been abandoned in favor of the EthPM V3 smart contract packaging standard defined in ERC-2678

Simple Summary

A data format describing a smart contract software package.

Abstract

This EIP defines a data format for *package manifest* documents, representing a package of one or more smart contracts, optionally including source code and any/all deployed instances across multiple networks. Package manifests are minified JSON objects, to be distributed via content addressable storage networks, such as IPFS.

This document presents a natural language description of a formal specification for version **2** of this format.

Motivation

This standard aims to encourage the Ethereum development ecosystem towards software best practices around code reuse. By defining an open, community-driven package data format standard, this effort seeks to provide support for package management tools development by offering a general-purpose solution that has been designed with observed common practices in mind.

As version 2 of this specification, this standard seeks to address a number of areas of improvement found for the previous version (defined in EIP-190). This version:

- Generalizes storage URIs to represent any content addressable URI scheme, not only IPFS.
- Renames *release lockfile* to *package manifest*.
- Adds support for languages other than Solidity by generalizing the compiler information format.
- Redefines link references to be more flexible, to represent arbitrary gaps in bytecode (besides only addresses), in a more straightforward way.
- Forces format strictness, requiring that package manifests contain no extraneous whitespace, and sort object keys in alphabetical order, to prevent hash mismatches.

Specification

This document defines the specification for an EthPM package manifest. A package manifest provides metadata about a Package, and in most cases should provide sufficient information about the packaged contracts and its dependencies to do bytecode verification of its contracts.

> ***Note***
>
> *A hosted version of this specification is available via GitHub Pages. This EIP and the hosted HTML document were both autogenerated from the same documentation source.*

Guiding Principles

This specification makes the following assumptions about the document lifecycle.

1. Package manifests are intended to be generated programmatically by package management software as part of the release process.
2. Package manifests will be consumed by package managers during tasks like installing package dependencies or building and deploying new releases.
3. Package manifests will typically **not** be stored alongside the source, but rather by package registries *or* referenced by package registries and stored in something akin to IPFS.

Conventions

RFC2119

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

- https://www.ietf.org/rfc/rfc2119.txt

Prefixed vs Unprefixed

A prefixed hexadecimal value begins with `0x`. Unprefixed values have no prefix. Unless otherwise specified, all hexadecimal values **should** be represented with the `0x` prefix.

Prefixed	`0xdeadbeef`
Unprefixed	`deadbeef`

Document Format

The canonical format is a single JSON object. Packages **must** conform to the following serialization rules.

- The document **must** be tightly packed, meaning no linebreaks or extra whitespace.
- The keys in all objects must be sorted alphabetically.
- Duplicate keys in the same object are invalid.
- The document **must** use UTF-8 encoding.
- The document **must** not have a trailing newline.

Document Specification

The following fields are defined for the package. Custom fields **may** be included. Custom fields **should** be prefixed with `x-` to prevent name collisions with future versions of the specification.

See Also	Formalized (JSON-Schema) version of this specification: package.spec.json
Jump To	Definitions

EthPM Manifest Version: `manifest_version`

The `manifest_version` field defines the specification version that this document conforms to. Packages **must** include this field.

Required	Yes
Key	`manifest_version`
Type	String
Allowed Values	`2`

Package Name: `package_name`

The `package_name` field defines a human readable name for this package. Packages **must** include this field. Package names **must** begin with a lowercase letter and be comprised of only lowercase letters, numeric characters, and the dash character `-`. Package names **must** not exceed 214 characters in length.

Required	Yes
Key	`package_name`
Type	String
Format	**must** match the regular expression `^[a-zA-Z][a-zA-Z0-9_]{0,255}$`

Package Meta: `meta`

The `meta` field defines a location for metadata about the package which is not integral in nature for package installation, but may be important or convenient to have on-hand for other reasons. This field **should** be included in all Packages.

Required	No
Key	`meta`
Type	Package Meta Object

Version: `version`

The `version` field declares the version number of this release. This value **must** be included in all Packages. This value **should** conform to the semver version numbering specification.

Required	Yes

Key	`version`
Type	String

Sources: `sources`

The `sources` field defines a source tree that **should** comprise the full source tree necessary to recompile the contracts contained in this release. Sources are declared in a key/value mapping.

Key	`sources`
Type	Object (String: String)
Format	See Below.

Format

Keys **must** be relative filesystem paths beginning with a `./`.

Paths **must** resolve to a path that is within the current working directory.

Values **must** conform to *one of* the following formats.

- Source string.
- Content Addressable URI.

When the value is a source string the key should be interpreted as a file path.

- If the resulting document is a directory the key should be interpreted as a directory path.
- If the resulting document is a file the key should be interpreted as a file path.

Contract Types: `contract_types`

The `contract_types` field holds the Contract Types which have been included in this release. Packages **should** only include contract types that can be found in the source files for this package. Packages **should not** include contract types from dependencies. Packages **should not** include abstract contracts in the contract types section of a release.

Key	`contract_types`
Type	Object (String: Contract Type Object)
Format	Keys **must** be valid Contract Aliases. Values **must** conform to the Contract Type Object definition.

Deployments: `deployments`

The `deployments` field holds the information for the chains on which this release has Contract Instances as well as the Contract Types and other deployment details for those deployed contract instances. The set of chains defined by the `*BIP122 URI <#bip122-uris>*` keys for this object **must** be unique.

Key	`deployments`
Type	Object (String: Object(String: Contract Instance Object))
Format	See Below.

Format

Keys **must** be a valid BIP122 URI chain definition.

Values **must** be objects which conform to the following format.

- Keys **must** be valid Contract Instance Names.
- Values **must** be a valid Contract Instance Object.

Build Dependencies: `build_dependencies`

The `build_dependencies` field defines a key/value mapping of Ethereum Packages that this project depends on.

Required	No
Key	`build_dependencies`
Type	Object (String: String)
Format	Keys **must** be valid package names matching the regular expression `[a-z][-a-z0-9]{0,213}`. Values **must** be valid IPFS URIs which resolve to a valid package.

Definitions

Definitions for different objects used within the Package. All objects allow custom fields to be included. Custom fields **should** be prefixed with `x-` to prevent name collisions with future versions of the specification.

The *Link Reference* Object

A Link Reference object has the following key/value pairs. All link references are assumed to be associated with some corresponding Bytecode.

Offsets: `offsets`

The `offsets` field is an array of integers, corresponding to each of the start positions where the link reference appears in the bytecode. Locations are 0-indexed from the beginning of the bytes representation of the corresponding bytecode. This field is invalid if it references a position that is beyond the end of the bytecode.

Required	Yes
Type	Array

Length: `length`

The `length` field is an integer which defines the length in bytes of the link reference. This field is invalid if the end of the defined link reference exceeds the end of the bytecode.

Required	Yes
Type	Integer

Name: `name`

The `name` field is a string which **must** be a valid Identifier. Any link references which **should** be linked with the same link value **should** be given the same name.

Required	No
Type	String
Format	**must** conform to the Identifier format.

The *Link Value* Object

Describes a single Link Value.

A **Link Value object** is defined to have the following key/value pairs.

Offsets: `offsets`

The `offsets` field defines the locations within the corresponding bytecode where the `value` for this link value was written. These locations are 0-indexed from the beginning of the bytes representation of the corresponding bytecode.

Required	Yes
Type	Integer
Format	See Below.

Format

Array of integers, where each integer **must** conform to all of the following.

- greater than or equal to zero
- strictly less than the length of the unprefixed hexadecimal representation of the corresponding bytecode.

Type: `type`

The `type` field defines the `value` type for determining what is encoded when linking the corresponding bytecode.

Required	Yes
Type	String
Allowed Values	`"literal"` for bytecode literals `"reference"` for named references to a particular Contract Instance

Value: `value`

The `value` field defines the value which should be written when linking the corresponding bytecode.

Required	Yes
Type	String
Format	Determined based on `type`, see below.

Format

For static value *literals* (e.g. address), value **must** be a *byte string*

To reference the address of a Contract Instance from the current package the value should be the name of that contract instance.

- This value **must** be a valid contract instance name.
- The chain definition under which the contract instance that this link value belongs to must contain this value within its keys.
- This value **may not** reference the same contract instance that this link value belongs to.

To reference a contract instance from a Package from somewhere within the dependency tree the value is constructed as follows.

- Let `[p1, p2, .. pn]` define a path down the dependency tree.
- Each of `p1, p2, pn` **must** be valid package names.
- `p1` **must** be present in keys of the `build_dependencies` for the current package.
- For every `pn` where `n > 1`, `pn` **must** be present in the keys of the `build_dependencies` of the package for `pn-1`.
- The value is represented by the string `<p1>:<p2>:<...>:<pn>:<contract-instance>` where all of `<p1>`, `<p2>`, `<pn>` are valid package names and `<contract-instance>` is a valid Contract Name.
- The `<contract-instance>` value **must** be a valid Contract Instance Name.
- Within the package of the dependency defined by `<pn>`, all of the following must be satisfiable:
  - There **must** be *exactly* one chain defined under the `deployments` key which matches the chain definition that this link value is nested under.
  - The `<contract-instance>` value **must** be present in the keys of the matching chain.

The *Bytecode* Object

A bytecode object has the following key/value pairs.

Bytecode: `bytecode`

The `bytecode` field is a string containing the `0x` prefixed hexadecimal representation of the bytecode.

Required	Yes

Type	String
Format	`0x` prefixed hexadecimal.

Link References: `link_references`

The `link_references` field defines the locations in the corresponding bytecode which require linking.

Required	No
Type	Array
Format	All values **must** be valid Link Reference objects. See also below.

Format

This field is considered invalid if *any* of the Link References are invalid when applied to the corresponding `bytecode` field, *or* if any of the link references intersect.

Intersection is defined as two link references which overlap.

Link Dependencies: `link_dependencies`

The `link_dependencies` defines the Link Values that have been used to link the corresponding bytecode.

Required	No
Type	Array
Format	All values **must** be valid Link Value objects. See also below.

Format

Validation of this field includes the following:

- Two link value objects **must not** contain any of the same values for `offsets`.
- Each link value object **must** have a corresponding link reference object under the `link_references` field.
- The length of the resolved `value` **must** be equal to the `length` of the corresponding Link Reference.

The *Package Meta* Object

The *Package Meta* object is defined to have the following key/value pairs.

Authors: `authors`

The `authors` field defines a list of human readable names for the authors of this package. Packages **may** include this field.

Required	No
Key	`authors`
Type	Array (String)

License: `license`

The `license` field declares the license under which this package is released. This value **should** conform to the SPDX format. Packages **should** include this field.

Required	No
Key	`license`
Type	String

Description: `description`

The `description` field provides additional detail that may be relevant for the package. Packages **may** include this field.

Required	No
Key	`description`
Type	String

Keywords: `keywords`

The `keywords` field provides relevant keywords related to this package.

Required	No
Key	`keywords`
Type	List of Strings

Links: `links`

The `links` field provides URIs to relevant resources associated with this package. When possible, authors **should** use the following keys for the following common resources.

- `website`: Primary website for the package.
- `documentation`: Package Documentation
- `repository`: Location of the project source code.

Key	`links`
Type	Object (String: String)

The *Contract Type* Object

A *Contract Type* object is defined to have the following key/value pairs.

Contract Name: `contract_name`

The `contract_name` field defines the Contract Name for this Contract Type.

Required	If the Contract Name and Contract Alias are not the same.
Type	String
Format	**must** be a valid Contract Name.

Deployment Bytecode: `deployment_bytecode`

The `deployment_bytecode` field defines the bytecode for this Contract Type.

Required	No
Type	Object
Format	**must** conform to the Bytecode Object format.

Runtime Bytecode: `runtime_bytecode`

The `runtime_bytecode` field defines the unlinked `0x` -prefixed runtime portion of Bytecode for this Contract Type.

Required	No
Type	Object
Format	**must** conform to the Bytecode Object format.

ABI: `abi`

Required	No
Type	List
Format	**must** conform to the Ethereum Contract ABI JSON format.

Natspec: `natspec`

Required	No
Type	Object
Format	The union of the UserDoc and DevDoc formats.

Compiler: `compiler`

Required	No
Type	Object
Format	**must** conform to the Compiler Information object format.

The *Contract Instance* Object

A **Contract Instance Object** represents a single deployed Contract Instance and is defined to have the following key/value pairs.

Contract Type: `contract_type`

The `contract_type` field defines the Contract Type for this Contract Instance. This can reference any of the contract types included in this Package *or* any of the contract types found in any of the package dependencies from the `build_dependencies` section of the Package Manifest.

Required	Yes
Type	String
Format	See Below.

Format

Values for this field **must** conform to *one of* the two formats herein.

To reference a contract type from this Package, use the format `<contract-alias>`.

- The `<contract-alias>` value **must** be a valid Contract Alias.
- The value **must** be present in the keys of the `contract_types` section of this Package.

To reference a contract type from a dependency, use the format `<package-name>:<contract-alias>`.

- The `<package-name>` value **must** be present in the keys of the `build_dependencies` of this Package.
- The `<contract-alias>` value **must** be be a valid Contract Alias.
- The resolved package for `<package-name>` must contain the `<contract-alias>` value in the keys of the `contract_types` section.

Address: `address`

The `address` field defines the Address of the Contract Instance.

Required	Yes
Type	String
Format	Hex encoded `0x` prefixed Ethereum address matching the regular expression

	`0x[0-9a-fA-F]{40}`.

Transaction: `transaction`

The `transaction` field defines the transaction hash in which this Contract Instance was created.

Required	No
Type	String
Format	`0x` prefixed hex encoded transaction hash.

Block: `block`

The `block` field defines the block hash in which this the transaction which created this *contract instance* was mined.

Required	No
Type	String
Format	`0x` prefixed hex encoded block hash.

Runtime Bytecode: `runtime_bytecode`

The `runtime_bytecode` field defines the runtime portion of bytecode for this Contract Instance. When present, the value from this field supersedes the `runtime_bytecode` from the Contract Type for this Contract Instance.

Required	No
Type	Object
Format	**must** conform to the Bytecode Object

	format.

Every entry in the `link_references` for this bytecode **must** have a corresponding entry in the `link_dependencies` section.

Compiler: `compiler`

The `compiler` field defines the compiler information that was used during compilation of this Contract Instance. This field **should** be present in all Contract Types which include `bytecode` or `runtime_bytecode`.

Required	No
Type	Object
Format	**must** conform to the Compiler Information Object format.

The *Compiler Information* Object

The `compiler` field defines the compiler information that was used during compilation of this Contract Instance. This field **should** be present in all contract instances that locally declare `runtime_bytecode`.

A *Compiler Information* object is defined to have the following key/value pairs.

Name `name`

The `name` field defines which compiler was used in compilation.

Required	Yes
Key	`name`
Type	String

Version: `version`

The `version` field defines the version of the compiler. The field **should** be OS agnostic (OS not included in the string) and take the form of either the stable version in semver format or if built on a nightly should be denoted in the form of `<semver>-<commit-hash>` ex: `0.4.8-commit.60cc1668`.

Required	Yes
Key	`version`
Type	String

Settings: `settings`

The `settings` field defines any settings or configuration that was used in compilation. For the `"solc"` compiler, this **should** conform to the Compiler Input and Output Description.

Required	No
Key	`settings`
Type	Object

BIP122 URIs

BIP122 URIs are used to define a blockchain via a subset of the BIP-122 spec.

```
blockchain://<genesis_hash>/block/<latest confirmed block hash>
```

The `<genesis hash>` represents the blockhash of the first block on the chain, and `<latest confirmed block hash>` represents the hash of the latest block that's been reliably confirmed (package managers should be free to choose their desired level of confirmations).

Rationale

The following use cases were considered during the creation of this specification.

owned	A package which contains contracts which are not meant to be used by themselves but rather as base contracts to provide functionality to other contracts through inheritance.
transferable	A package which has a single dependency.
standard-token	A package which contains a reusable contract.
safe-math-lib	A package which contains deployed instance of one of the package contracts.
piper-coin	A package which contains a deployed instance of a reusable contract from a dependency.
escrow	A package which contains a deployed instance of a local contract which is linked against a deployed instance of a local library.
wallet	A package with a deployed instance of a local contract which is linked against a deployed instance of a library from a dependency.
wallet-with-send	A package with a deployed instance which links against a deep dependency.

Each use case builds incrementally on the previous one.

A full listing of Use Cases can be found on the hosted version of this specification.

Glossary

ABI

The JSON representation of the application binary interface. See the official specification for more information.

Address

A public identifier for an account on a particular chain

Bytecode

The set of EVM instructions as produced by a compiler. Unless otherwise specified this should be assumed to be hexadecimal encoded, representing a whole number of bytes, and prefixed with `0x`.

Bytecode can either be linked or unlinked. (see Linking)

Unlinked Bytecode	The hexadecimal representation of a contract's EVM instructions that contains sections of code that requires linking for the contract to be functional. The sections of code which are unlinked **must** be filled in with zero bytes. **Example**: `0x606060405260e06000730000000000000000000000000000000000000000634d536f`
Linked Bytecode	The hexadecimal representation of a contract's EVM instructions which has had all Link References replaced with the desired Link Values. **Example**: `0x606060405260e06000736fe36000604051602001526040518160e060020a634d536f`

Chain Definition

This definition originates from BIP122 URI.

A URI in the format `blockchain://<chain_id>/block/<block_hash>`

- `chain_id` is the unprefixed hexadecimal representation of the genesis hash for the chain.

- `block_hash` is the unprefixed hexadecimal representation of the hash of a block on the chain.

A chain is considered to match a chain definition if the the genesis block hash matches the `chain_id` and the block defined by `block_hash` can be found on that chain. It is possible for multiple chains to match a single URI, in which case all chains are considered valid matches

Content Addressable URI

Any URI which contains a cryptographic hash which can be used to verify the integrity of the content found at the URI.

The URI format is defined in RFC3986

It is **recommended** that tools support IPFS and Swarm.

Contract Alias

This is a name used to reference a specific Contract Type. Contract aliases **must** be unique within a single Package.

The contract alias **must** use *one of* the following naming schemes:

- `<contract-name>`
- `<contract-name>[<identifier>]`

The `<contract-name>` portion **must** be the same as the Contract Name for this contract type.

The `[<identifier>]` portion **must** match the regular expression `\[[-a-zA-Z0-9]{1,256}]`.

Contract Instance

A contract instance a specific deployed version of a Contract Type.

All contract instances have an Address on some specific chain.

Contract Instance Name

A name which refers to a specific Contract Instance on a specific chain from the deployments of a single Package. This name **must** be unique across all other contract instances for the given chain. The name must conform to the regular expression `[a-zA-Z][a-zA-Z0-9_]{0,255}`

In cases where there is a single deployed instance of a given Contract Type, package managers **should** use the Contract Alias for that contract type for this name.

In cases where there are multiple deployed instances of a given contract type, package managers **should** use a name which provides some added semantic information as to help differentiate the two deployed instances in a meaningful way.

Contract Name

The name found in the source code that defines a specific Contract Type. These names **must** conform to the regular expression `[a-zA-Z][-a-zA-Z0-9_]{0,255}`.

There can be multiple contracts with the same contract name in a projects source files.

Contract Type

Refers to a specific contract in the package source. This term can be used to refer to an abstract contract, a normal contract, or a library. Two contracts are of the same contract type if they have the same bytecode.

Example:

```
contract Wallet {
    ...
}
```

A deployed instance of the `Wallet` contract would be of of type `Wallet`.

Identifier

Refers generally to a named entity in the Package.

A string matching the regular expression `[a-zA-Z][-_a-zA-Z0-9]{0,255}`

Link Reference

A location within a contract’s bytecode which needs to be linked. A link reference has the following properties.

`offset`	Defines the location within the bytecode where the link reference begins.
`length`	Defines the length of the reference.
`name`	(optional.) A string to identify the reference

Link Value

A link value is the value which can be inserted in place of a Link Reference

Linking

The act of replacing Link References with Link Values within some Bytecode.

Package

Distribution of an application's source or compiled bytecode along with metadata related to authorship, license, versioning, et al.

For brevity, the term **Package** is often used metonymously to mean Package Manifest.

Package Manifest

A machine-readable description of a package (See Specification for information about the format for package manifests.)

Prefixed

Bytecode string with leading `0x`.

Example	`0xdeadbeef`

Unprefixed

Not Prefixed.

Example	`deadbeef`

Backwards Compatibility

This specification supports backwards compatibility by use of the manifest_version property. This specification corresponds to version `2` as the value for that field.

Implementations

This submission aims to coincide with development efforts towards widespread implementation in commonly-used development tools.

The following tools are known to have begun or are nearing completion of a supporting implementation.

- Truffle
- Populus
- Embark

Full support in implementation **may** require Further Work, specified below.

Further Work

This EIP addresses only the data format for package descriptions. Excluded from the scope of this specification are:

- Package registry interface definition
- Tooling integration, or how packages are stored on disk.

These efforts **should** be considered separate, warranting future dependent EIP submssions.

Acknowledgements

The authors of this document would like to thank the original authors of EIP-190, ETHPrize for their funding support, all community contributors, and the Ethereum community at large.

Copyright

Citation

Please cite this document as:

g. nicholas d'andrea, Piper Merriam, Nick Gheorghita, Danny Ryan, "EIP-1123: Revised Ethereum Smart Contract Packaging Standard," *Ethereum Improvement Proposals*, no. 1123, June 2018. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-1123.

Ethereum Improvement Proposals

Ethereum Improvement Proposals

ethereum/EIPs

Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards.