Ethereum Contract ABI

Edit New Page

chriseth edited this page 24 days ago · 64 revisions

Functions

Basic design

We assume the Application Binary Interface (ABI) is strongly typed, known at compilation time and static. No introspection mechanism will be provided. We assert that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time. Should these cases become important they can be adequately handled as facilities built within the Ethereum ecosystem.

Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

Types

The following elementary types exist:

- `uint<M>` : unsigned integer type of `M` bits, `0 < M <= 256` , `M % 8 == 0` . e.g. `uint32` , `uint8` , `uint256` .
- `int<M>` : two's complement signed integer type of `M` bits, `0 < M <= 256` , `M % 8 == 0` .
- `address` : equivalent to `uint160` , except for the assumed interpretation and language typing.
- `uint` , `int` : synonyms for `uint256` , `int256` respectively (not to be used for computing the function selector).
- `bool` : equivalent to `uint8` restricted to the values 0 and 1
- `fixed<M>x<N>` : fixed-point signed number of `M+N` bits, `0 < M + N <= 256` , `M % 8 == N % 8 == 0` . Corresponds to the int256 equivalent binary value divided by `2^M` .
- `ufixed<M>x<N>` : unsigned variant of `fixed<M>x<N>` .
- `fixed` , `ufixed` : synonyms for `fixed128x128` , `ufixed128x128` respectively (not to be used for computing the function selector).
- `bytes<M>` : binary type of `M` bytes, `0 < M <= 32` .
- `function` : equivalent to `bytes24` : an address, followed by a function selector

The following (fixed-size) array type exists:

- `<type>[M]` : a fixed-length array of the given fixed-length type.

The following non-fixed-size types exist:

- `bytes` : dynamic sized byte sequence.
- `string` : dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]` : a variable-length array of the given fixed-length type.

Formal Specification of the Encoding

We will now formally specify the encoding, such that it will have the following properties, which are especially useful if some arguments are nested arrays:

Properties:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]` . In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses"

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called "dynamic":

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k > 0`

All other types are called "static".

Definition: `len(a)` is the number of bytes in a binary string `a` . The type of `len(a)` is assumed to be `uint256` .

We define `enc` , the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition: For any ABI value `X` , we recursively define `enc(X)` , depending on the type of `X` being

- `T[k]` for any `T` and `k` :

  `enc(X) = head(X[0]) ... head(X[k-1]) tail(X[0]) ... tail(X[k-1])`

  where `head` and `tail` are defined for `X[i]` being of a static type as `head(X[i]) = enc(X[i])` and `tail(X[i]) = ""` (the empty string) and as `head(X[i]) = enc(len(head(X[0]) ... head(X[k-1]) tail(X[0]) ... tail(X[i-1])))` `tail(X[i]) = enc(X[i])` otherwise.

  Note that in the dynamic case, `head(X[i])` is well-defined since the lengths of the head parts only depend on the types and not the values. Its value is the offset of the beginning of `tail(X[i])` relative to the start of `enc(X)` .

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc([X[1], ..., X[k]])`

  i.e. it is encoded as if it were an array of static size `k` , prefixed with the number of elements.

ÐΞV Technologies

- RLP Encoding
- RLPx Node Discovery Protocol
- ÐΞVp2p Wire Protocol
- ÐΞVp2p Whitepaper (WiP)
- Web3 Secret Storage

Ethereum Technologies

- Patricia Tree
- Wire protocol
- Light client protocol
- Subtleties
- Solidity, Docs & ABI
- NatSpec Format
- Contract ABI
- Bad Block Reporting
- Bad Chain Canary
- Extra Data
- Brain Wallet

Ethash/Dashimoto

- Ethash
- Ethash C API
- Ethash DAG

Whisper

- Whisper Proposal
- Whisper Overview
- PoC-1 Wire protocol
- PoC-2 Wire protocol
- PoC-2 Whitepaper

Misc

- Hard Problems of Cryptocurrency
- Chain Fibers
- Glossary

Clone this wiki locally

https://github.com/ethere

Clone in Desktop

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.

- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is a multiple of 32 bytes.

- `address`: as in the `uint160` case
- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-oder (left) side with `0xff` for negative `X` and with zero bytes for positive `X` such that the length is a multiple of 32 bytes.
- `bool`: as in the `uint8` case, where `1` is used for `true` and `0` for `false`
- `fixed<M>x<N>`: `enc(X)` is `enc(X * 2**N)` where `X * 2**N` is interpreted as a `int256`.
- `fixed`: as in the `fixed128x128` case
- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 2**N)` where `X * 2**N` is interpreted as a `uint256`.
- `ufixed`: as in the `ufixed128x128` case
- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with zero-bytes to a length of 32.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

`function_selector(f) enc([a_1, ..., a_n])`

and the return values `v_1, ..., v_k` of `f` are encoded as

`enc([v_1, ..., v_k])`

where the types of `[a_1, ..., a_n]` and `[v_1, ..., v_k]` are assumed to be fixed-size arrays of length `n` and `k`, respectively. Note that strictly, `[a_1, ..., a_n]` can be an "array" with elements of different types, but the encoding is still well-defined as the assumed common type `T` (above) is not actually used.

Examples

Given the contract:

```
contract Foo {
  function bar(fixed[2] xy) {}
  function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }
  function sam(bytes name, bool z, uint[] data) {}
}
```

Thus for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into:

- `0xcdcd77c0`: the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045`: the first

parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter - boolean `true`, padded to 32 bytes

In total:

```
0xcdcd77c00000000000000000000000000000000000000000000000000000000000000045000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `[2.125, 8.5]`, we would pass 68 bytes total, broken down into:

- `0xab55044d` : the Method ID. This is derived from the signature `bar(fixed128x128[2])`. Note that `fixed` is replaced with its canonical representation `fixed128x128`.
- `0x0000000000000000000000000000000220000000000000000000000000000000` : the first part of the first parameter, a fixed128x128 value `2.125`.
- `0x0000000000000000000000000000000880000000000000000000000000000000` : the second part of the first parameter, a fixed128x128 value `8.5`.

In total:

```
0xab55044d00000000000000000000000000000002200000000000000000000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into:

- `0xa5643bf2` : the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060` : the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0` : the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004` : the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000` : the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002` : the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the third entry of the third parameter.

In total:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000
```

Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c642100000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

```
0x8be65246
0000000000000000000000000000000000000000000000000000000000000123
0000000000000000000000000000000000000000000000000000000000000080
3132333435363738393000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000e0
0000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000456
0000000000000000000000000000000000000000000000000000000000000789
000000000000000000000000000000000000000000000000000000000000000d
48656c6c6f2c20776f726c642100000000000000000000000000000000000000
```

Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to

3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which as not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address` : the address of the contract (intrinsically provided by Ethereum);
- `topics[0]` : `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo` , it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated;
- `topics[n]` : `EVENT_INDEXED_ARGS[n - 1]` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data` : `abi_serialise(EVENT_NON_INDEXED_ARGS)` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_serialise` is the ABI serialisation function used for returning a series of typed values from a function, as described above).

JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields:

- `type` : `"function"` or `"constructor"` (can be omitted, defaulting to function);
- `name` : the name of the function (only present for function types);
- `inputs` : an array of objects, each of which contains:
  - `name` : the name of the parameter;
  - `type` : the canonical type of the parameter.
- `outputs` : an array of objects similar to `inputs` , can be omitted.

An event description is a JSON object with fairly similar fields:

- `type` : always `"event"`
- `name` : the name of the event;
- `inputs` : an array of objects, each of which contains:
  - `name` : the name of the parameter;
  - `type` : the canonical type of the parameter.
  - `indexed` : `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous` : `true` if the event was declared as `anonymous` .

For example,

```
contract Test {
function Test(){ b = 0x12345678901234567890123456789012; }
event Event(uint indexed a, bytes32 b)
event Event2(uint indexed a, bytes32 b)
function foo(uint a) { Event(a, b); }
bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes
"name":"Event2"
```

```
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Example Javascript Usage

```
var Test = eth.contract(
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]);
var theTest = new Test(addrTest);

// examples of usage:
// every log entry ("event") coming from theTest (i.e. Event & Event2):
var f0 = eth.filter(theTest);
// just log entries ("events") of type "Event" coming from theTest:
var f1 = eth.filter(theTest.Event);
// also written as
var f1 = theTest.Event();
// just log entries ("events") of type "Event" and "Event2" coming from theTest:
var f2 = eth.filter([theTest.Event, theTest.Event2]);
// just log entries ("events") of type "Event" coming from theTest with indexed p
var f3 = eth.filter(theTest.Event, {'a': 69});
// also written as
var f3 = theTest.Event({'a': 69});
// just log entries ("events") of type "Event" coming from theTest with indexed p
var f4 = eth.filter(theTest.Event, {'a': [69, 42]});
// also written as
var f4 = theTest.Event({'a': [69, 42]});

// options may also be supplied as a second parameter with `earliest`, `latest`,
var options = { 'max': 100 };
var f4 = theTest.Event({'a': [69, 42]}, options);

var trigger;
f4.watch(trigger);

// call foo to make an Event:
theTest.foo(69);

// would call trigger like:
//trigger(theTest.Event, {'a': 69, 'b': '0x12345678901234567890123456789012'}, n)
// where n is the block number that the event triggered in.
```

Implementation:

```
// e.g. f4 would be similar to:
web3.eth.filter({'max': 100, 'address': theTest.address, 'topics': [ [69, 42] ]})
```

```
// except that the resultant data would need to be converted from the basic log e
{
  'address': theTest.address,
  'topics': [web3.sha3("Event(uint256,bytes32)"), 0x00...0045 /* 69 in hex format
  'data': '0x12345678901234567890123456789012',
  'number': n
}
// into data good for the trigger, specifically the three fields:
  Test.Event // derivable from the first topic
  {'a': 69, 'b': '0x12345678901234567890123456789012'} // derivable from the 'ind
  n // from the 'number'
```

Event result:

```
[ {
  'event': Test.Event,
  'args': {'a': 69, 'b': '0x12345678901234567890123456789012'},
  'number': n
  },
  { ...
  } ...
]
```

JUST DONE! [develop branch]

NOTE: THIS IS OLD - IGNORE IT unless reading for historical purposes

- Internal LogFilter, log-entry matching mechanism and eth_installFilter needs to support matching multiple values (OR semantics) *per* topic index (at present it will only match topics with AND semantics and set-inclusion, not per-index).

i.e. at present you can only ask for each of a number of given topic values to be matched throughout each topic:

- `topics: [69, 42, "Gav"]` would match against logs with 3 topics `[42, 69, "Gav"]`, `["Gav", 69, 42]` but **not** against logs with topics `[42, 70, "Gav"]`.

we need to be able to provide one of a number of topic values, and, each of these options for each topic index:

- `topics: [[69, 42], [] /* anything */, "Gav"]` should match against logs with 3 topics `[42, 69, "Gav"]`, `[42, 70, "Gav"]` but **not** against `["Gav", 69, 42]`.

[English | Deutsch | Español | Français | 日本語 | Română | فارسی | Italiano | 한국어 | 中文]