title: Simple serialize description: Explanation of Ethereum's SSZ format. lang: en sidebarDepth: 2

Simple serialize (SSZ) is the serialization method used on the Beacon Chain. It replaces the RLP serialization used on the execution layer everywhere across the consensus layer except the peer discovery protocol. SSZ is designed to be deterministic and also to Merkleize efficiently. SSZ can be thought of as having two components: a serialization scheme and a Merkleization scheme that is designed to work efficiently with the serialized data structure.

How does SSZ work? {#how-does-ssz-work}

Serialization {#serialization}

SSZ is a serialization scheme that is not self-describing - rather it relies on a schema that must be known in advance. The goal of SSZ serialization is to represent objects of arbitrary complexity as strings of bytes. This is a very simple process for "basic types". The element is simply converted to hexadecimal bytes. Basic types include:

- unsigned integers
- Booleans

For complex "composite" types, serialization is more complicated because the composite type contains multiple elements that might have different types or different sizes, or both. Where these objects all have fixed lengths (i.e. the size of the elements is always going to be constant irrespective of their actual values) the serialization is simply a conversion of each element in the composite type ordered into little-endian bytestrings. These bytestrings are joined together. The serialized object has the bytelist representation of the fixed-length elements in the same order as they appear in the deserialized object.

For types with variable lengths, the actual data gets replaced by an "offset" value in that element's position in the serialized object. The actual data gets added to a heap at the end of the serialized object. The offset value is the index for the start of the actual data in the heap, acting as a pointer to the relevant bytes.

The example below illustrates how the offsetting works for a container with both fixed and variable-length elements:

```Rust

```
struct Dummy {

    number1: u64,
    number2: u64,
    vector: Vec<u8>,
    number3: u64
}

dummy = Dummy{

    number1: 37,
    number2: 55,
    vector: vec![1,2,3,4],
    number3: 22,
}

serialized = ssz.serialize(dummy)
```

```

`serialized` would have the following structure (only padded to 4 bits here, padded to 32 bits in reality, and keeping the `int` representation for clarity):

``` [37, 0, 0, 0, 55, 0, 0, 0, 16, 0, 0, 0, 22, 0, 0, 0, 1, 2, 3, 4]

---

```
   | | | | |
```

number1 number2 offset for number 3 value for vector vector
```

```

divided over lines for clarity:

```
[37, 0, 0, 0, # little-endian encoding of `number1`. 55, 0, 0, 0, # little-endian encoding of `number2`. 16, 0,
0, 0, # The "offset" that indicates where the value of `vector` starts (little-endian 16). 22, 0, 0, 0, # little-
endian encoding of `number3`. 1, 2, 3, 4, # The actual values in `vector`.]
```

This is still a simplification - the integers and zeros in the schematics above would actually be stored bytelists, like this:

```
[ 10100101000000000000000000000000 # little-endian encoding of `number1` 10110111000000000000000000000000 #
little-endian encoding of `number2`. 10010000000000000000000000000000 # The "offset" that indicates where the
value of `vector` starts (little-endian 16). 10010110000000000000000000000000 # little-endian encoding of
`number3`. 1000000110000001010000001110000100 # The actual value of the `bytes` field. ]
```

So the actual values for variable-length types are stored in a heap at the end of the serialized object with their offsets stored in their correct positions in the ordered list of fields.

There are also some special cases that require specific treatment, such as the`BitList` type that requires a length cap to be added during serialization and removed during deserialization. Full details are available in the SSZ spec.

Deserialization {#deserialization}

To deserialize this object requires the **schema**. The schema defines the precise layout of the serialized data so that each specific element can be deserialized from a blob of bytes into some meaningful object with the elements having the right type, value, size and position. It is the schema that tells the deserializer which values are actual values and which ones are offsets. All field names disappear when an object is serialized, but reinstantiated on deserialization according to the schema.

See ssz.dev for an interactive explainer on this.

Merkleization {#merkleization}

This SSZ serialized object can then be merkleized - that is transformed into a Merkle-tree representation of the same data. First, the number of 32-byte chunks in the serialized object is determined. These are the "leaves" of the tree. The total number of leaves must be a power of 2 so that hashing together the leaves eventually produces a single hash-tree-root. If this is not naturally the case, additional leaves containing 32 bytes of zeros are added. Diagrammatically:

```
hash tree root / \ / \ / \ / \ hash of leaves hash of leaves 1 and 2 3 and 4 / \ / \ / \ / \ / \ / \ leaf1
leaf2 leaf3 leaf4

There are also cases where the leaves of the tree do not naturally evenly distribute in the way they do in the example above. For example, leaf 4 could be a container with multiple elements that require additional "depth" to be added to the Merkle tree, creating an uneven tree.

Instead of referring to these tree elements as leaf X, node X etc, we can give them generalized indices, starting with root = 1 and counting from left to right along each level. This is the generalized index explained above. Each element in the serialized list has a generalized index equal to `2**depth + idx` where idx is its zero-indexed position in the serialized object and the depth is the number of levels in the Merkle tree, which can be determined as the base-two logarithm of the number of elements (leaves).

## Generalized indices {#generalized-indices}

A generalized index is an integer that represents a node in a binary Merkle tree where each node has a generalized index`2 ** depth + index in row`.

``` 1 --depth = 0 2**0 + 0 = 1 2 3 --depth = 1 2**1 + 0 = 2, 2**1+1 = 3 4 5 6 7 --depth = 2 2**2 + 0 = 4, 2**2 + 1 = 5...

```

This representation yields a node index for each piece of data in the Merkle tree.

Multiproofs {#multiproofs}

Providing the list of generalized indices representing a specific element allows us to verify it against the hash-tree-root. This root is our accepted version of reality. Any data we are provided can be verified against that reality by inserting it into the right place in the Merkle tree (determined by its generalized index) and observing that the root remains constant. There are functions in the spec [here]() that show how to compute the minimal set of nodes required to verify the contents of a particular set of generalized indices.

For example, to verify data in index 9 in the tree below, we need the hash of the data at indices 8, 9, 5, 3, 1. The hash of (8,9) should equal hash (4), which hashes with 5 to produce 2, which hashes with 3 to produce the tree root 1. If incorrect data was provided for 9, the root would change - we would detect this and fail to verify the branch.

``` * = data required to generate proof

```
                    1*
        2   3*
4   5*   6   7
```

8 *9* 10 11 12 13 14 15

```

Further reading {#further-reading}

- [Upgrading Ethereum: SSZ]()
- [Upgrading Ethereum: Merkleization]()
- [SSZ implementations]()
- [SSZ calculator]()
- [SSZ.dev]()