Polkadot Runtime Environment
Protocol Specification

April 18, 2019

1 Conventions and Definitions

Definition 1. ***Runtime*** *is the state transition function of the decentralized ledger protocol.*

Definition 2. *A* ***path graph*** *or a* ***path*** *of n nodes formally referred to as $\boldsymbol{P_n}$, is a tree with two nodes of vertex degree 1 and the other n-2 nodes of vertex degree 2. Therefore, P_n can be represented by sequences of $(v_1, \ldots, v_n)$ where $e_i = (v_i, v_{i+1})$ for $1 \leqslant i \leqslant n - 1$ is the edge which connect v_i and v_{i+1}.*

Definition 3. ***Radix-r tree*** *is a variant of a trie in which:*

- *Every node has at most r children where $r = 2^x$ for some x;*
- *Each node that is the only child of a parent, which does not represent a valid key is merged with its parent.*

As a result, in a radix tree, any path whose interior vertices all have only one child and does not represent a valid key in the data set, is compressed into a single edge. This improves space efficiency when the key space is sparse.

Definition 4. *By a* ***sequences of bytes*** *or a* ***byte array****, b, of length n, we refer to*

$$b := (b_0, b_1, \ldots, b_{n-1}) \text{ such that } 0 \leqslant b_i \leqslant 255$$

We define $\mathbb{B}_n$ to be the ***set of all byte arrays of length n****. Furthermore, we define:*

$$\mathbb{B} := \bigcup_{i=0}^{\infty} \mathbb{B}_i$$

Notation 5. *We represent the concatenation of byte arrays $a := (a_0, \ldots, a_n)$ and $b := (b_0, \ldots, b_m)$ by:*

$$a || b := (a_0, \ldots, a_n, b_0, \ldots, b_m)$$

Definition 6. *For a given byte b the* ***bitwise representation*** *of b is defined as*

$$b := b^7 \ldots b^0$$

where

$$b = 2^0 b^0 + 2^1 b^1 + \cdots + 2^7 b^7$$

Definition 7. *By the* ***little-endian*** *representation of a non-negative integer, I, represented as*

$$I = (B_n \ldots B_0)_{256}$$

in base 256, we refer to a byte array $B = (b_0, b_1, \ldots, b_n)$ such that

$$b_i := B_i$$

Accordingly, define the function $\mathrm{Enc}_{\mathrm{LE}}$*:*

$$\begin{array}{rlcl} \mathrm{Enc}_{\mathrm{LE}}: & \mathbb{Z}^+ & \rightarrow & \mathbb{B} \\ & (B_n \ldots B_0)_{256} & \mapsto & (B_0, B_1, \ldots, B_n) \end{array}$$

Definition 8. *By* **`UINT32`** *we refer to a non-negative integer stored in a byte array of length 4 using little-endian encoding format.*

Definition 9. *A* ***blockchain*** *C is a directed path graph. Each node of the graph is called* ***Block*** *and indicated by* $\boldsymbol{B}$*. The unique sink of C is called* ***Genesis Block****, and the source is called the* ***Head*** *of C. For any vertex (B_1, B_2) where $B_1 \rightarrow B_2$ we say B_2 is the* ***parent*** *of B_1 and we indicate it by*

$$B_2 := P(B_1)$$

2 Block

In Polkadot RE, a block is made of two main parts, namely the *block header* and the *list of extrinsics*. *The Extrinsics* represent the generalization of the concept of *transaction*, containing any set of data that is external to the system, and which the underlying chain wishes to validate and keep track of.

2.1 Block Header

The block header is designed to be minimalistic in order to boost the efficiency of the light clients. It is defined formally as follows:

Definition 10. *The* ***header of block B****,* **Head($\boldsymbol{B}$)** *is a 5-tuple containing the following elements:*

- ***parent_hash:*** *is the 32-byte Blake2s hash of the header of the parent of the block indicated henceforth by* $\boldsymbol{H_p}$*.*
- ***number:*** *formally indicated as* $\boldsymbol{H_i}$ *is an integer, which represents the index of the current block in the chain. It is equal to the number of the ancestor blocks. The genesis block has number 0.*
- ***state_root:*** *formally indicated as* $\boldsymbol{H_r}$ *is the root of the Merkle trie, whose leaves implement the storage for the system.*
- ***extrinsics_root:*** *is the field which is reserved for the runtime to validate the integrity of the extrinsics composing the block body. For example, it can hold the root hash of the Merkle trie which stores an ordered list of the extrinsics being validated in this block. This element is formally referred to as* $\boldsymbol{H_e}$*.*
- ***digest:*** *this field is used to store any chain-specific auxiliary data, which could help the light clients interact with the block without the need of accessing the full storage. Polkadot RE does not impose any limitation or specification for this field. Essentially, it can be a byte array of any length. This field is indicated as* $\boldsymbol{H_d}$

Definition 11. *The **Block Header Hash of Block B, $H_h(b)$**, is the hash of the header of block B encoded by simple codec:*

$$H_b(b) := \text{Blake2}\, s(\text{Enc}_{\text{SC}}(\text{Head}(B)))$$

2.2 Justified Block Header

The Justified Block Header is provided by the consensus engine and presented to the Polkadot RE, for the block to be appended to the blockchain. It contains the following parts:

- **block_header** the complete block header as defined in Section 2.1 and denoted by Head(B).
- **justification**: as defined by the consensus specification indicated by Just(B) [link this to its definition from consensus].
- **authority Ids**: This is the list of the Ids of authorities, which have voted for the block to be stored and is formally referred to as $A(B)$. An authority Id is 32bit.

2.3 Extrinsics

Each block contains as well a list of extrinsics. Polkadot RE does not specify or limit the internal of each extrinsics beside the fact that each extrinsics is a byte array encoded using SCALE codec [46].

The extrinsics_root is set by the runtime, and its value is opaque to Polkadot RE.

The extrinsics in a block are ordered using pairing each extrinsics by a UINT32 integer sequential number starting at 0 which is encoded using SCALE codec.

2.4 Block Format

3 Interactions with the Runtime

Runtime is the code implementing the logic of the chain. This code is decoupled from the Polkadot RE to make the Runtime easily upgradable without the need to upgrade the Polkadot RE itself. In this section, we describe the details upon which the Polkadot RE is interacting with the Runtime.

3.1 Loading the Runtime code

Polkadot RE expects to receive the code for the runtime of the chain as a compiled WebAssembly (Wasm) Blob. The current runtime is stored in the state database under the key represented as a byte array:

$$b := \text{3A,63,6F,64,65}$$

which is the byte array of ASCII representation of string ":code" (see Section 11). For any call to the runtime, Polkadot RE makes sure that it has the most updated Runtime as calls to runtime have potentially the ability to change the runtime code.

The initial runtime code of the chain is embedded as an extrinsics into the chain initialization JSON file and is submitted to Polkadot RE (see Section 10).

Subsequent calls to the runtime have the ability to call the storage API (see Section A) to insert a new Wasm blob into runtime storage slot to upgrade the runtime.

3.2 Code Executor

Polkadot RE provides a Wasm Virtual Machine (VM) to run the Runtime. The Wasm VM exposes the Polkadot RE API to the Runtime, which, on its turn, executes a call to the Runtime entries stored in the Wasm module. This part of the Runtime environment is referred to as the ***Executor***.

In this section, we specify the general setup for an Executor call into the Runtime. In Section 3.3 we specify the parameters and the return values of each Runtime entry separately.

3.2.1 ABI Encoding between Runtime and the Runtime Environment

All data exchanged between Polkadot RE and the Runtime is encoded using SCALE codec described in Section 9.1.

3.2.2 Access to Runtime API

When Polkadot RE calls a Runtime entry it should make sure Runtime has access to the all Polkadot Runtime API functions described in Appendix A. This can be done for example by loading another Wasm module alongside the runtime which imports these functions from Polkadot RE as host functions.

3.2.3 Sending Arguments to Runtime

In each invocation of a Runtime entry, the arguments which are supposed to be sent to the entry need to be encoded using SCALE codec into a byte array B. The Executor then needs to retrieve the memory buffer of the Runtime Wasm module and extend it to fit the size of the byte array. Then it needs to copy the byte array value in the correct offset of the extended buffer. Finally, when the Wasm method corresponding to the entry is called, two UINT32 integers are sent to the method as arguments. The first one is the offset of the byte array B in the extended shared memory buffer, and the second one is the size of B.

3.2.4 The Return Value from a Runtime Entry

The value which is returned from the invocation represents two consecutive UINT32 integers in which the first one indicates the pointer to the offset of the result returned by the entry encoded in SCALE codec in the memory buffer. The second one provides the size of the blob.

3.3 Entries into Runtime

Polkadot RE assumes that at least the following functions are implemented in the Runtime Wasm blob and has been exported as shown in Snippet 1:

```
(export "Core_version" (func $Core_version))
(export "Core_authorities" (func $Core_authorities))
(export "Core_execute_block" (func $Core_execute_block))
(export "Core_initialise_block" (func $Core_initialise_block))
(export "Metadata_metadata" (func $Metadata_metadata))
(export "BlockBuilder_apply_extrinsic" (func $BlockBuilder_apply_extrinsic))
(export "BlockBuilder_finalise_block" (func $BlockBuilder_finalise_block))
(export "BlockBuilder_inherent_extrinsics"
        (func $BlockBuilder_inherent_extrinsics))
(export "BlockBuilder_check_inherents" (func $BlockBuilder_check_inherents))
(export "BlockBuilder_random_seed" (func $BlockBuilder_random_seed))
(export "TaggedTransactionQueue_validate_transaction"
        (func $TaggedTransactionQueue_validate_transaction))
(export "OffchainWorkerApi_offchain_worker"
        (func $OffchainWorkerApi_offchain_worker))
(export "ParachainHost_duty_roster" (func $ParachainHost_duty_roster))
(export "ParachainHost_active_parachains"
        (func $ParachainHost_active_parachains))
(export "ParachainHost_parachain_head" (func $ParachainHost_parachain_head))
(export "ParachainHost_parachain_code" (func $ParachainHost_parachain_code))
(export "GrandpaApi_grandpa_pending_change"
        (func $GrandpaApi_grandpa_pending_change))
(export "GrandpaApi_grandpa_forced_change"
        (func $GrandpaApi_grandpa_forced_change))
(export "GrandpaApi_grandpa_authorities"
        (func $GrandpaApi_grandpa_authorities))
(export "ParachainHost_validators" (func $Core_authorities))
```

Snippet 1. Snippet to export entries into tho Wasm runtime module

The following sections describe the standard based on which Polkadot RE communicates with each runtime entry.

3.3.1 `Core_version`

This entry receives no argument; it returns the version data encoded in ABI format described in Section 3.2.1 containing the following data:

Name	Type	Description
`spec_name`	String	runtime identifier
`impl_name`	String	the name of the implementation (e.g. C++)
`authoring_version`	UINT32	the version of the authorship interface
`spec_version`	UINT32	the version of the runtime specification
`impl_version`	UINT32	the version of the runtime implementation
`apis`	ApisVec	List of supported AP

Table 1. Detail of the version data type returns from runtime `version` function

3.3.2 `Core_authorities`

This entry is to report the set of authorities at a given block. It receives `block_id` as an argument; it returns an array of `authority_id`'s.

3.3.3 `Core_execute_block`

This entry is responsible for executing all extrinsics in the block and reporting back the changes into the state storage. It receives the block header and the block body as its arguments, and it returns a triplet:

Name	Type	Description
`results`	Boolean	Indicating if the execution was su
`storage_changes`	[???]	Contains all changes to the state storage
`change_updat`	[???]	

Table 2. Detail of the data execute_block returns after execution

3.3.4 `Core_initialise_block`

3.3.5 `TaggedTransactionQueue_validate_transaction`

[Explain function]

4 Network Interactions

4.1 Extrinsics Submission

Extrinsic submission is made by sending an extrinsic network message. The structure of this message is specified in Definition 12.

Upon receiving an extrinsics message, Polkadot RE decodes the transaction and calls `validate_trasaction` runtime function defined in Section 3.3.5, to check the validity of the extrinsic. If `validate_transaction` considers the submitted extrinsics as a valid one, Polkadot RE makes the extrinsics available for the consensus engine for inclusion in future blocks.

4.2 Network Messages

Definition 12. ***Extrinsic submission network message: [Extrinsic submission network message definition]***

4.3 Block Submission and Validation

Block validation is the process, by which the client asserts that a block is fit to be added to the blockchain. This means that the block is consistent with the world state and transitions from the state of the system to a new valid state.

Blocks can be handed to the Polkadot RE both from the network stack and from the consensus engine.

Both the Runtime and the Polkadot RE need to work together to assure block validity. This can be accomplished by Polkadot RE invoking `execute_block` entry into the runtime as a part of the validation process.

Polkadot RE implements the following procedure to assure the validity of the block:

Algorithm 1. Import-and-Validate-Block(B, Just(B))

```
1: Verify-Block-Justification($B$, Just($B$))
2: if $B$ is Finalized and $P(B)$ is not Finalized
3:     Mark-as-Final($P(B)$)
4: Verify $H_{p(B)} \in$ Blockchain
5: State-Changes = Runtime($B$)
6: Update-World-State(State-Changes)
```

For the definition of the finality and the finalized block see Section 7.3.

5 State Storage and the Storage Trie

For storing the state of the system, Polkadot RE implements a hash table storage where the keys are used to access each data entry. There is no limitation neither on the size of the key nor on the size of the data stored under them, besides the fact that they are byte arrays.

5.1 Accessing The System Storage

Polkadot RE implements various functions to facilitate access to the system storage for the runtime. Section A lists all of those functions. Here we formalize the access to the storage when it is being directly accessed by Polkadot RE (in contrast to Polkadot runtime).

Definition 13. *The **StoredValue** function retrieves the value stored under a specific key in the state storage and is formally defined as :*

$$\begin{array}{rl} \text{StoredValue:} & \mathbb{B} \rightarrow \mathbb{B} \\ & k \mapsto \begin{cases} v & \textit{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases} \end{array}$$

5.2 The General Tree Structure

In order to ensure the integrity of the state of the system, the stored data needs to be re-arranged and hashed in a *modified Merkle Patricia Tree*, which hereafter we refer to as the ***Trie***. This rearrangment is necessary to be able to compute the Merkle hash of the whole or part of the state storage, consistently and efficiently at any given time.

The Trie is used to compute the *state root*, H_r, (see Definition 10), whose purpose is to authenticate the validity of the state database. Thus, Polkadot RE follows a rigorous encoding algorithm to compute the values stored in the trie nodes to ensure that the computed Merkle hash, H_r, matches across the Polkadot RE implementations.

The Trie is a *radix-16* tree as defined in Definition 3. Each key value identifies a unique node in the tree. However, a node in a tree might or might not be associated with a key in the storage.

When traversing the Trie to a specific node, its key can be reconstructed by concatenating the subsequences of the key which are stored either explicitly in the nodes on the path or implicitly in their position as a child of their parent.

To identify the node corresponding to a key value, k, first we need to encode k in a consistent with the Trie structure way. Because each node in the trie has at most 16 children, we represent the key as a sequence of 4-bit nibbles:

Definition 14. *For the purpose of labeling the branches of the Trie, the key k is encoded to k_{enc} using KeyEncode functions:*

$$k_{\text{enc}} := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) := \text{KeyEncode}(k) \tag{1}$$

such that:

$$\text{KeyEncode}(k) \colon \begin{cases} \mathbb{B} & \rightarrow \text{Nibbles}^4 \\ k := (b_1, \ldots, b_n) := & \mapsto (b_1^1, b_1^2, b_2^1, b_2^2, \ldots, b_n^1, b_n^2) \\ & := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) \end{cases}$$

where Nibble^4 *is the set of all nibbles of 4-bit arrays and b_i^1 and b_i^2 are 4-bit nibbles, which are the big endian representations of b_i:*

$$(b_i^1, b_i^2) := (b_i / 16, b_i \operatorname{mod} 16)$$

, where mod is the remainder and / is the integer division operators.

By looking at k_{enc} as a sequence of nibbles, one can walk the radix tree to reach the node identifying the storage value of k.

5.3 The Trie structure

In this subsection, we specify the structure of the nodes in the Trie as well as the Trie structure:

Notation 15. *We refer to the* ***set of the nodes of Polkadot state trie*** *by $\mathcal{N}$. By $N \in \mathcal{N}$ to refer to an individual node in the trie.*

Definition 16. *The State Trie is a radix-16 tree. Each Node in the Trie is identified with a unique key k_N such that:*

- *k_N is the shared prefix of the key of all the descendants of N in the Trie.*

and, at least one of the following statements holds:

- *(k_N, v) corresponds to an existing entry in the State Storage.*
- *N has more than one child.*

Conversely, if (k, v) is an entry in the State Trie then there is a node $N \in \mathcal{N}$ such that $k_N = k$.

Notation 17. *A* ***branch*** *node is a node which has one child or more. A branch node can have at most 16 children. A* ***leaf*** *node is a childless node. Accordingly:*

$$\mathcal{N}_b := \{N \in \mathcal{N} \mid N \text{ is a branch node}\}$$
$$\mathcal{N}_l := \{N \in \mathcal{N} \mid N \text{ is a leaf node}\}$$

For each Node, part of k_N is built while the trie is traversed from root to N part of k_N is stored in N as formalized in Definition 18.

Definition 18. *For any $N \in \mathcal{N}$, its key k_N is divided into an* ***aggregated prefix key***, $\mathbf{pk}_N^{\mathbf{Agr}}$*, aggregated by Algorithm 2 and a* ***partial key***, $\mathbf{pk}_N$ *of length $0 \leqslant l_{\mathrm{pk}_N} \leqslant 65535$ such that:*

$$\mathrm{pk}_N := (k_{\mathrm{enc}_i}, \ldots, k_{\mathrm{enc}_{i+l_{\mathrm{pk}_N}}})$$

where pk_N is a suffix subsequence of k_N; and we have:

$$\mathrm{KeyEncode}(k_N) = \mathrm{pk}_N^{\mathrm{Agr}} | \mathrm{pk}_N = (k_{\mathrm{enc}_1}, \ldots, k_{\mathrm{enc}_{i-1}}, k_{\mathrm{enc}_i}, k_{\mathrm{enc}_{i+l_{\mathrm{pk}_N}}})$$

Part of $\mathrm{pk}_N^{\mathrm{Agr}}$ is explicitly stored in N's ancestors. Additionally, for each ancestor, a single nibble is implicitly derived while traversing from the ancestor to its child included in the traversal path using the Index_N function defined in Definition 19.

Definition 19. *For $N \in \mathcal{N}_b$ and N_c child of N, we define* $\mathbf{Index}_N$ *function as:*

$$\begin{aligned} \mathrm{Index}_N \colon \ \{N_c \in \mathcal{N} \mid N_c \textit{ is a child of } N\} &\to \mathrm{Nibbles}_1^4 \\ N_c &\mapsto i \end{aligned}$$

such that

$$k_{N_c} = k_N || i || \mathrm{pk}_{N_c}$$

Assuming that P_N is the path (see Definition 2) from the Trie root to node N, Algorithm 2 rigorously demonstrates how to build $\mathrm{pk}_N^{\mathrm{Agr}}$ while traversing P_N.

Algorithm 2. Aggregate-Key(P_N: =(TrieRoot $= N_1, \ldots, N_j = N$))

```
1:  $\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \phi$
2:  $i \leftarrow 1$
3:  **while** $(N_i \neq N)$
4:      $\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \mathrm{pk}_N^{\mathrm{Agr}} || \mathrm{pk}_N$
5:      $\mathrm{pk}_N^{\mathrm{Agr}} \leftarrow \mathrm{pk}_N^{\mathrm{Agr}} || \mathrm{Index}_{N_i}(N_{i+1})$
6:  **return** $\mathrm{pk}_N^{\mathrm{Agr}}$
```

Definition 20. *A node $N \in \mathcal{N}$ stores the* ***node value***, $\boldsymbol{v_N}$*, which consists of the following concatenated data:*

Node Header	Partial key	Node Subvalue

Formally noted as:

$$v_N := \mathrm{Head}_N || \mathrm{Enc}_{\mathrm{HE}}(\mathrm{pk}_N) || \mathrm{sv}_N$$

where Head_N, pk_N, $\mathrm{Enc}_{\mathrm{nibbles}}$ and sv_N are defined in Definitions 21,18, ? and 23, respectively.

Definition 21. *The* ***node header*** *of node N, Head_N, consists of $l > 1$ bytes*

Node Type	pk length	pk length extra byte 1	pk key length extra byte 2	…	pk length extra byte l
$\mathrm{Head}_{N,1}^{6-7}$	$\mathrm{Head}_{N,1}^{0-5}$	$\mathrm{Head}_{N,2}$	....		$\mathrm{Head}_{N,l+1}$

In which $\mathrm{Head}_{N,1}^{6-7}$*, the two most significant bits of the first byte of* Head_N *are determined as follows:*

$$\mathrm{Head}_{N,1}^{6-7} := \begin{cases} 00 & \text{Special case} \\ 01 & \text{Leaf Node} \\ 10 & \text{Branch Node with } k_N \notin \mathcal{K} \\ 11 & \text{Branch Node with } k_N \in \mathcal{K} \end{cases}$$

and $\mathrm{Head}_{N,1}^{0-5}$*, the 6 least significant bits of the first byte of* Head_N *are defined to be:*

$$\mathrm{Head}_{N,1}^{0-5} := \begin{cases} \|\mathrm{pk}_N\|_{\mathrm{nib}} & \|\mathrm{pk}_N\|_{\mathrm{nib}} < 63 \\ 63 & \|\mathrm{pk}_N\|_{\mathrm{nib}} \geqslant 63 \end{cases}$$

In which $\|\mathbf{pk}_{\boldsymbol{N}}\|_{\mathbf{nib}}$ *is the length of* pk_N *in number nibbles.* $\mathrm{Head}_{N,2}, \ldots, \mathrm{Head}_{N,l}$ *bytes are determined by Algorithm 3.*

Algorithm 3. Partial-Key-Length-Encoding($\mathrm{Head}_{N,1}^{6-7}, \mathrm{pk}_N$)

```
1:  if $\|\mathrm{pk}_N\|_{\mathrm{nib}} \geqslant 2^{16}$
2:      return Error
3:  $\mathrm{Head}_{N,1} \leftarrow 64 \times \mathrm{Head}_{N,1}^{6-7}$
4:  if $\|\mathrm{pk}_N\|_{\mathrm{nib}} < 63$
5:      $\mathrm{Head}_{N,1} \leftarrow \mathrm{Head}_{N,1} + \mathrm{pk}_N$
6:      return $\mathrm{Head}_N$
7:  $\mathrm{Head}_{N,1} \leftarrow \mathrm{Head}_{N,1} + 63$
8:  $l \leftarrow \|\mathrm{pk}_N\|_{\mathrm{nib}} - 62$
9:  $i \leftarrow 2$
10: while ($l > 255$)
11:     $\mathrm{Head}_{N,i} \leftarrow 255$
12:     $l \leftarrow l - 255$
13:     $i \leftarrow i + 1$
14: $\mathrm{Head}_{N,i} \leftarrow l - 1$
```

5.4 The Merkle proof

To prove the consistency of the state storage across the network and its modifications both efficiently and effectively, the Trie implements a Merkle tree structure. The hash value corresponding to each node needs to be computed rigorously to make the inter-implementation data integrity possible.

The Merkle value of each node should depend on the Merkle value of all its children as well as on its corresponding data in the state storage. This recursive dependancy is encompassed into the subvalue part of the node value which recursively depends on the Merkle value of its children.

We use the auxilary function introduced in Definition ? to encode and decode information stored in a branch node.

Definition 22. *Suppose $N_b, N_c \in \mathcal{N}$ and N_c is a child of N_b. We define where bit $b_i := 1$ if N has a child with partial key i, therefore we define* ***ChildrenBitmap*** *functions as follows:*

$$\begin{aligned}\text{ChildrenBitmap}\colon\ & \mathcal{N}_b \rightarrow \mathbb{B}_2\\ & N \mapsto (b_{15}, \ldots, b_8, b_7, \ldots b_0)_2\end{aligned}$$

where

$$b_i := \begin{cases} 1 & \exists N_c \in \mathcal{N}\colon k_{N_c} = k_{N_b} || i || \text{pk}_{N_c}\\ 0 & \textit{otherwise}\end{cases}$$

Definition 23. *For a given node N, the* ***subvalue*** *of N, formally referred to as sv_N, is determined as follows: in a case which:*

$$\text{sv}_N := \begin{cases} \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & N \textit{ is a leaf node}\\ \text{ChildrenBitmap}(N) \| H(N_{C_1}) \ldots H(N_{C_n}) || \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & N \textit{ is a branch node}\end{cases}$$

Where $N_{C_1} \ldots N_{C_n}$ with $n \leqslant 16$ are the children nodes of the branch node N and Enc_{SC}, StoredValue, H, and ChildrenBitmap(N) are defined in Definitions 46,13, 24 and 22 respectively.

The Trie deviates from a traditional Merkle tree where node value, v_N (see Definition 20) is presented instead of its hash if it occupies less space than its hash.

Definition 24. *For a given node N, the* ***Merkle value*** *of N, denoted by $H(N)$ is defined as follows:*

$$H\colon \mathbb{B} \rightarrow \mathbb{B}_{32}$$
$$H(N)\colon \begin{cases} v_N || 0_{B_{32 - \| v_N \|}} & \| v_N \| < 32\\ \text{Blake2}s(v_N) & \| v_N \| \geqslant 32\end{cases}$$

Where $0_{32 - \| v_N \|}$ an all zero byte array of length $32 - || v_N ||$.

6 Transactions

6.1 Preliminaries

Definition 25. ***Account key*** $(\mathbf{sk}^a, \mathbf{pk}^a)$ *is a pair of Ristretto SR25519 used to sign transactions among other accounts and blance-related functions.*

7 Consensus Engine

Consensus in Polkadot RE is achieved during the execution of two different procedures. The first procedure is block production and the second is finality. Polkadot RE must run these procedures, if and only if it is running on a validator node.

7.1 Block Tree

In the course of formation of a (distributed) blockchain, it is possible that the chain forks into multiple subchains in various block positions. We refer to this structure as a *block tree:*

Definition 26. *The **Block Tree** of a blockchain is the union of all different versions of the blockchain observed by all the nodes in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is a parent of B_2.*

Because every block in the blockchain contains a reference to its parent, it is easy to see that the block tree is actually a tree.

A block tree naturally imposes partial order relationships on the blocks as follows:

Definition 27. *We say **B is descendant of B'**, formally noted as **$B > B'$** if B is a descendant of B' in the block tree.*

7.2 Block Production

Polkadot RE uses BABE protocol [Gro19] for block production designed based on Ouroboros praos [DGKR18]. BABE execution happens in sequential non-overlapping phases known as an ***epoch***. Each epoch on its turn is divided into a predefined number of slots. All slots in each epoch are sequentially indexed starting from 0. At the beginning of each epoch, the BABE node needs to run Algorithm ? to find out in which slots it should produce a block and gossip to the other block producers. In turn, the block producer node should keep a copy of the block tree and grow it as it receives valid blocks from other block producers. A block producer prunes the tree in parallel using Algorithm ?.

7.2.1 Preliminaries

Definition 28. *A **Block Producer**, noted by $\mathcal{P}_j$ is a node running Polkadot RE which is authorized to keep a transaction queue and which gets a turn in producing blocks.*

Definition 29. ***Block signing key pair** $(\mathbf{sk}_j^s, \mathbf{pk}_j^s)$ is an ED25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (see Definition 25) and is used to sign the produced block. Similarly **block lottery key pair**, noted by $(\mathbf{sk}_j^v, \mathbf{pk}_j^s)$, is a ED25519 key pair which is used by $\mathcal{P}_j$ to compute its lottery values.*

Definition 30. *A block production **epoch**, formally refered to as $\mathcal{E}$ is a period with pre-known starting time and fixed length during which the set of block producers stays constant. Epochs are indexed sequentially, we refer to the n^{th} epoch since genesis by $\mathcal{E}_n$. Each epoch is divided into equal length periods known as block production **slot**s sequentially indexed in each epoch. Each slot is awarded to a subset of block producers during which they are allowed to generate a block.*

Notation 31. *We refer to the number of slots in epoch $\mathcal{E}_n$ by sc_n. sc_n is set to ??? at the genesis.*

7.2.2 Block Production Lottery

Definition 32. ***Winning threshold*** *denoted by* $\boldsymbol{\tau}$ *is the threshold which is used in Algorirthm ? to decide if a block producer is the winner of a specific slot.* τ *is initially set to ???.*

During each epoch, each block producer node should run Algorithm ? to identify the slots it is awarded. These are the slots during which the block producer is allowed to build a block. The sk^v is the block producer lottery secret key and n is the epoch for whose slots the block producer is running the lottery.

Algorithm 4. BLOCK-PRODUCTION-LOTTERY(sk^v, n)

```
1: $r \leftarrow$ EPOCH-RANDOMNESS($n$)
2: for $i := 1$ to $\mathrm{sc}_n$
3:     $(d, \pi) \leftarrow \mathrm{VRF}(i, \mathrm{sk}^v)$
4:     $A[i] \leftarrow (i, d, \pi)$
5: return A
```

For any slot i in epoch n where $d < \tau$, the block producer is required to produce a block. For the definitions of EPOCH-RANDOMNESS and VRF functions, see Sections 8.1 and 8.2 respectively.

7.2.3 Block Production

7.3 Finality

Polkadot RE uses GRANDPA Finality protocol [Ali19] to finalize blocks. Finality is obtained by consecutive rounds of voting by validator nodes. Validators execute GRANDPA finality process in parallel to Block Production as an independent service. In this section, we describe the different functions that GRANDPA service is supposed to perform to successfully participate in the block finalization process.

7.3.1 Preliminaries

Definition 33. *A* ***GRANDPA Voter****,* v*, is represented by a key pair* $(k_v^{\mathrm{pr}}, v_{\mathrm{id}})$ *where* k_v^{pr} *represents its private key which is an* ED25519 *private key, is a node running GRANDPA protocol, and broadcasts votes to finalize blocks in a Polkadot RE - based chain. The* ***set of all GRANDPA voters*** *is indicated by* $\mathbb{V}$*. For a given block B, we have4*

$$\mathbb{V}_B = \texttt{authorities}(B)$$

where `authorities` *is the entry into runtime described in Section 3.3.2.*

Definition 34. ***GRANDPA state****,* GS*, is defined as*

$$\mathrm{GS} := \{\mathbb{V}, \mathrm{id}_{\mathbb{V}}, r\}$$

where:

$\mathbb{V}$*: is the set of voters.*

$\mathbb{V}_{\mathbf{id}}$*: is an incremental counter tracking membership, which changes in V.*

$\boldsymbol{r}$*: is the voting round number.*

Now we need to define how Polkadot RE counts the number of votes for block B. First a vote is defined as:

Definition 35. *A **GRANDPA vote** or simply a vote for block B is an ordered pair defined as*

$$V(B) := (H_h(B), H_i(B))$$

where $H_h(B)$ and $H_i(B)$ are the block hash and the block number defined in Definitions 10 and 11 respectively.

Definition 36. *Voters engage in a maximum of two sub-rounds of voting for each round r. The first sub-round is called **pre-vote** and the second sub-round is called **pre-commit**.*

By $\boldsymbol{V_v^{r,\mathrm{pv}}}$ and $\boldsymbol{V_v^{r,\mathrm{pc}}}$ we refer to the vote cast by voter v in round r (for block B) during the pre-vote and the pre-commit sub-round respectively.

The GRANDPA protocol dictates how an honest voter should vote in each sub-round, which is described in Algorithm 6. After defining what constitues a vote in GRANDPA, we define how GRANDPA counts votes.

Definition 37. *Voter v **equivocates** if they broadcast two or more valid votes to blocks not residing on the same branch of the block tree during one voting sub-round. In such a situation, we say that v is an **equivocator** and any vote $V_v^{r,\mathrm{stage}}(B)$ cast by v in that round is an **equivocatory vote** and*

$$\mathcal{E}^{r,\mathrm{stage}}$$

represents the set of all equivocators voters in sub-round "stage" of round r. When we want to refer to the number of equivocators whose equivocation has been observed by voter v we refer to it by:

$$\mathcal{E}_{\mathrm{obs}(v)}^{r,\mathrm{stage}}$$

Definition 38. *A vote $V_v^{r,\mathrm{stage}} = V(B)$ is **invalid** if*

- *$H(B)$ does not correspond to a valid block;*
- *B is not an (eventual) descendant of a previously finalized block;*
- *$M_v^{r,\mathrm{stage}}$ does not bear a valid signature;*
- *$\mathrm{id}_{\mathbb{V}}$ does not match the current $\mathbb{V}$;*
- *If $V_v^{r,\mathrm{stage}}$ is an equivocatory vote.*

Definition 39. *For validator v, **the set of observed direct votes for Block B in round r**, formally denoted by $\mathrm{VD}_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B)$ is equal to the union of:*

- *set of valid votes $V_{v_i}^{r,\mathrm{stage}}$ cast in round r and received by v such that $V_{v_i}^{r,\mathrm{stage}} = V(B)$.*

Definition 40. *We refer to **the set of total votes observed by voter v in sub-round "stage" of round r** by $\boldsymbol{V_{\mathrm{obs}(v)}^{r,\mathrm{stage}}}$.*

*The **set of all observed votes by v in the sub-round stage of round r for block B**, $\boldsymbol{V_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B)}$ is equal to all of the observed direct votes casted for block B and all of the B's descendents defined formally as:*

$$V_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B) := \bigcup_{v_i \in \mathbb{V}, B \geqslant B'} \mathrm{VD}_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B')$$

The ***total number of observed votes for Block B in round r*** *is defined to be the size of that set plus the total number of equivocators voters:*

$$\#V_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B) = |V_{\mathrm{obs}(v)}^{r,\mathrm{stage}}(B)| + |\mathcal{E}_{\mathrm{obs}(v)}^{r,\mathrm{stage}}|$$

Definition 41. *The current* ***pre-voted*** *block $B_v^{r,\mathrm{pv}}$ is the block with*

$$H_n(B_v^{r,\mathrm{pv}}) = \mathrm{Max}(H_n(B) | \forall B \colon \#V_{\mathrm{obs}(v)}^{r,\mathrm{pv}}(B) \geqslant 2/3|\mathbb{V}|)$$

Note that for genesis block Genesis we always have $\#V_{\mathrm{obs}(v)}^{r,\mathrm{pv}}(B) = |\mathbb{V}|$.

Finally, we define when a voter v see a round as completable, that is when they are confident that $B_v^{r,\mathrm{pv}}$ is an upper bound for what is going to be finalised in this round.

Definition 42. *We say that round r is* ***completable*** *if $|V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}| + \mathcal{E}_{\mathrm{obs}(v)}^{r,\mathrm{pc}} > \frac{2}{3}\mathbb{V}$ and for all $B' > B_v^{r,\mathrm{pv}}$:*

$$|V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}| - \mathcal{E}_{\mathrm{obs}(v)}^{r,\mathrm{pc}} - |V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}(B')| > \frac{2}{3}|\mathbb{V}|$$

Note that in practice we only need to check the inequality for those $B' > B_v^{r,\mathrm{pv}}$ where $|V_{\mathrm{obs}(v)}^{r,\mathrm{pc}}(B')| > 0$.

7.3.2 Voting Messages Specification

Voting is done by means of broadcasting voting messages to the network. Validators inform their peers about the block finalized in round r by broadcasting a finalization message (see Algorithm 6 for more details). These messages are specified in this section.

Definition 43. *A vote casted by voter v should be broadcasted as a* ***message $M_v^{r,\mathrm{stage}}$*** *to the network by voter v with the following structure:*

$$M_v^{r,\mathrm{stage}} := \mathrm{Enc}_{\mathrm{SC}}(r, \mathrm{id}_{\mathbb{V}}, \mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, \mathrm{Sig}_{\mathrm{ED25519}}(\mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, r, V_{\mathrm{id}}), v_{\mathrm{id}})$$

Where:

r:	*round number*	*64 bit integer*
V_{id}:	*incremental change tracker counter*	*64 bit integer*
v_{id}:	*Ed25519 public key of v*	*4 byte array*
stage:	*0 if it is the pre-vote sub-round*	*1 byte*
	1 if it the pre-commit sub-round	

Definition 44. *The* ***justification for block B in round r*** *of GRANDPA protocol defined $J^r(B)$ is a vector of pairs of the type:*

$$(V(B'), (\mathrm{Sign}_{v_i}^{r,\mathrm{pc}}(B'), v_{\mathrm{id}}))$$

in which either

$$B' > B$$

or $V_{v_i}^{r,\mathrm{pc}}(B')$ is an equivocatory vote.

In all cases, $\mathrm{Sign}_{v_i}^{r,\mathrm{pc}}(B')$ *is the signature of voter* v_i *broadcasted during the pre-commit sub-round of round* r*.*

Definition 45. GRANDPA *finalizing message for block B in round r* *represented as* $\boldsymbol{M_v^{r,\mathrm{Fin}}(B)}$ *is a message broadcasted by voter* v *to the network indicating that voter* v *has finalized block* B *in round* r*. It has the following structure:*

$$M_v^{r,\mathrm{Fin}}(B) := \mathrm{Enc}_{\mathrm{SC}}(r, V(B), J^r(B))$$

in which $J^r(B)$ *in the justification defined in Definition 44.*

7.3.3 Initiating the GRANDPA State

A validator needs to initiate its state and sync it with other validators, to be able to participate coherently in the voting process. In particular, considering that voting is happening in different rounds and each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r in accordance with the current voting round r_n, which is currently undergoing in the network.

As instructed in Algorithm 5, whenever the membership of GRANDPA voters changes, r is set to 0 and V_{id} needs to be incremented.

Algorithm 5. JOIN-LEAVE-GRANDPA-VOTERS ($\mathcal{V}$)

```
1: $r \leftarrow 0$
2: $\mathcal{V}_{\mathrm{id}} \leftarrow \mathrm{ReadState}('\mathrm{AUTHORITY\_SET\_KEY}')$
3: $\mathcal{V}_{\mathrm{id}} \leftarrow \mathcal{V}_{\mathrm{id}} + 1$
4: EXECUTE-ONE-GRANDPA-ROUND($r$)
```

Each voter should run Algorithm ? to verify that a round is completable.

7.3.4 Voting Process in Round *r*

For each round r, an honest voter v must participate in the voting process by following Algorithm 6.

Algorithm 6. PLAY-GRANDPA-ROUND(r)

```
1: $t_{r,v} \leftarrow \mathrm{Time}$
2: primary ← DERIVE-PRIMARY
3: if $v = \mathrm{primary}$
4:     BROADCAST($M_v^{r-1,\mathrm{Fin}}$(BEST-FINAL-CANDIDATE($r$-1)))
5: RECEIVE-MESSAGES(until Time $\geqslant t_{r,v} + 2 \times T$ or $r$ is completable)
6: $L \leftarrow$ BEST-FINAL-CANDIDATE($r$-1)
7: if RECEIVED($M_{v_{\mathrm{primary}}}^{r,\mathrm{pv}}(B)$) and $B_v^{r,\mathrm{pv}} \geqslant B > L$
8:     $N \leftarrow B$
9: else
10:    $N \leftarrow B' : H_n(B') = \max\{H_n(B') : B' > L\}$
11: BROADCAST($M_v^{r,\mathrm{pv}}(N)$)
12: RECEIVE-MESSAGES(until $B_v^{r,\mathrm{pv}} \geqslant L$ and (Time $\geqslant t_{r,v} + 4 \times T$ or $r$ is completable))
```

```
13: Broadcast($M_v^{r,\text{pc}}(B_v^{r,\text{pv}})$)
14: Play-Grandpa-round($r+1$)
```

Algorithm 7. Best-Final-Candidate(r)

```
1: $\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,\text{pv}} : |V_v^{r,\text{pc}}| - \#V_v^{r,\text{pc}}(B') \leqslant 1/3|\mathbb{V}|\}$
2: if $\mathcal{C} = \phi$
3:     return $\phi$
4: else
5:     return $E \in \mathcal{C} : H_n(E) = \max\{H_n(B') : B' \in \mathcal{C}\}$
```

Algorithm 8. Attempt-To-Finalize-Round(r)

```
1: $L \leftarrow$ Last-Finalized-Block
2: $E \leftarrow$ Best-Final-Candidate($r$)
3: if $E \geqslant L$ and $V_{\text{obs}(v)}^{r-1,\text{pc}}(E) > 2/3|\mathcal{V}|$
4:     Last-Finalized-Block$\leftarrow B^{r,\text{pc}}$
5:     if $M_v^{r,\text{Fin}}(E) \notin$ Received-Messages
6:         Broadcast($M_v^{r,\text{Fin}}(E)$)
7:         return
8: schedule-call Attempt-To-Finalize-Round($r$) when Receive-Messages
```

8 Cryptographic Algorithms

8.1 Randomness

8.2 VRF

9 Auxiliary Encodings

9.1 SCALE Codec

Polkadot RE uses *Simple Concatenated Aggregate Little-Endian" (SCALE) codec* to encode byte arrays that provide canonical encoding and to produce consistent hash values across their implementation, including the Merkle hash proof for the State Storage.

Definition 46. *The* ***SCALE codec*** *for* ***Byte array*** *A such that*

$$A := b_1\, b_2 \ldots b_n$$

such that $n < 2^{536}$ is a byte array refered to $\mathrm{Enc}_{\mathrm{SC}}(A)$ *and defined as follows:*

$$\mathrm{Enc}_{\mathrm{SC}}(A) := \begin{cases} l_1\, b_1\, b_2 \ldots b_n & 0 \leqslant n < 2^6 \\ i_1\, i_2\, b_1 \ldots b_n & 2^6 \leqslant n < 2^{14} \\ j_1\, j_2\, j_3\, b_1 \ldots b_n & 2^{14} \leqslant n < 2^{30} \\ k_1\, k_2 \ldots k_m\, b_1 \ldots b_n & 2^{30} \leqslant n \end{cases}$$

in which:

$l_1^1\, l_1^0 = 00$
$i_1^1\, i_1^0 = 01$
$j_1^1\, j_1^0 = 10$
$k_1^1\, k_1^0 = 11$

and n is stored in $\mathrm{Enc}_{\mathrm{SC}}(A)$ *in little-endian format in base-2 as follows:*

$$\left.\begin{array}{ll} l_1^7 \ldots l_1^3\, l_1^2 & n < 2^6 \\ i_2^7 \ldots i_2^0\, i_1^7 \ldots i_1^2 & 2^6 \leqslant n < 2^{14} \\ j_4^7 \ldots j_4^0\, j_3^7 \ldots j_1^7 \ldots j_1^2 & 2^{14} \leqslant n < 2^{30} \\ k_2 + k_3\, 2^8 + k_4\, 2^{2\cdot 8} + \cdots + k_m\, 2^{(m-2)8} & 2^{30} \leqslant n \end{array}\right\} := n$$

where:

$$k_1^7 \ldots k_1^3\, k_1^2 \colon = m - 4$$

Definition 47. *The* ***SCALE codec*** *for* ***Tuple*** *T such that:*

$$T := (A_1, \ldots, A_n)$$

Where A_i's are values of different types, is defined as:

$$\mathrm{Enc}_{\mathrm{SC}}(T) := \mathrm{Enc}_{\mathrm{SC}}(A_1) | \mathrm{Enc}_{\mathrm{SC}}(A_2) | \ldots | \mathrm{Enc}_{\mathrm{SC}}(A_n)$$

In case of a tuple (or struct), the knowledge of the shape of data is necessary for decoding.

9.2 Hex Encoding

Practically, it is more convenient and efficient to store and process data which is stored in a byte array. On the other hand, the Trie keys are broken into 4-bits nibbles. Accordingly, we need a method to encode sequences of 4-bits nibbles into byte arrays canonically:

Definition 48. *Suppose that* $\mathrm{PK} = (k_1, \ldots, k_n)$ *is a sequence of nibbles, then*
$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) :=$

$$\begin{cases} \mathrm{Nibbles}_4 & \rightarrow \mathbb{B} \\ \mathrm{PK} = (k_1, \ldots, k_n) & \mapsto \begin{cases} (16k_1 + k_2, \ldots, 16k_{2i-1} + k_{2i}) & n = 2\, i \\ (k_1, 16k_2 + k_3, \ldots, 16k_{2i} + k_{2i+1}) & n = 2\, i + 1 \end{cases} \end{cases}$$

10 Genesis Block Specification

11 Predefined Storage keys

12 Runtime upgrade

Appendix A Runtime API

Runtime API is a set of functions that Polkadot RE exposes to Runtime to access external functions needed for various reasons, such as Storage of content, access and manipulation, memory allocation, and also efficiency. The functions are specified in each subsequent subsection for each category of those functions.

A.1 Storage

A.1.1 `ext_set_storage`

Sets the value of a specific key in the state storage.

Prototype:

```
(func $ext_storage
  (param $key_data i32) (param $key_len i32) (param $value_data i32)
  (param $value_len i32))
```

Arguments:

- `key`: a pointer indicating the buffer containing the key.
- `key_len`: the key length in bytes.
- `value`: a pointer indicating the buffer containing the value to be stored under the key.
- `value_len`: the length of the value buffer in bytes.

A.1.2 `ext_storage_root`

Retrieves the root of the state storage.

Prototype:

```
(func $ext_storage_root
  (param $result_ptr i32))
```

Arguments:

- `result_ptr`: a memory address pointing at a byte array which contains the root of the state storage after the function concludes.

A.1.3 `ext_blake2_256_enumerated_trie_root`

Given an array of byte arrays, arranges them in a Merkle trie, defined in Section 5.4, and computes the trie root hash.

Prototype:

```
(func $ext_blake2_256_enumerated_trie_root
      (param $values_data i32) (param $lens_data i32) (param $lens_len i32)
      (param $result i32))
```

Arguments:

- `values_data`: a memory address pointing at the buffer containing the array where byte arrays are stored consecutively.
- `lens_data`: an array of `i32` elements each stores the length of each byte array stored in `value_data`.
- lens_len: the number of `i32` elements in `lens_data`.
- `result`: a memory address pointing at the beginning of a 32-byte byte array containing the root of the Merkle trie corresponding to elements of `values_data`.

A.1.4 `ext_clear_prefix`

Given a byte array, this function removes all storage entries whose key matches the prefix specified in the array.

Prototype:

```
(func $ext_clear_prefix
      (param $prefix_data i32) (param $prefix_len i32))
```

Arguments:

- `prefix_data`: a memory address pointing at the buffer containing the byte array containing the prefix.
- `prefix_len`: the length of the byte array in number of bytes.

A.1.5 `ext_clear_storage`

Given a byte array, this function removes the storage entry whose key is specified in the array.

Prototype:

```
(func $ext_clear_storage
      (param $key_data i32) (param $key_len i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.

A.1.6 `ext_exists_storage`

Given a byte array, this function checks if the storage entry corresponding to the key specified in the array exists.

Prototype:
```
(func $ext_exists_storage
      (param $key_data i32) (param $key_len i32) (result i32)
    )
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `result`: An `i32` integer which is equal to 1 verifies if an entry with the given key exists in the storage or 0 if the key storage does not contain an entry with the given key.

A.1.7 `ext_get_allocated_storage`

Given a byte array, this function allocates a large enough buffer in the memory and retrieves the value stored under the key that is specified in the array. Then, it stores it in the allocated buffer if the entry exists in the storage.

Prototype:
```
    (func $get_allocated_storage
      (param $key_data i32) (param $key_len i32) (param $written_out i32) (result i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `written_out`: the function stores the length of the retrieved value in number of bytes if the enty exists. If the entry does not exist, it returns $2^{32} - 1$.
- `result`: A pointer to the buffer in which the function allocates and stores the value corresponding to the given key if such an entry exist; otherwise it is equal to 0.

A.1.8 `ext_get_storage_into`

Given a byte array, this function retrieves the value stored under the key specified in the array and stores a specified chunk of it in the provided buffer, if the entry exists in the storage.

Prototype:
```
    (func $ext_get_storage_into
      (param $key_data i32) (param $key_len i32) (param $value_data i32)
      (param $value_len i32) (param $value_offset i32) (result i32))
```

Arguments:

- `key_data`: a memory address pointing at the buffer containing the byte array containing the key value.
- `key_len`: the length of the byte array in number of bytes.
- `value_data`: a pointer to the buffer in which the function stores the chunk of the value it retrieves.

- `value_len`: the (maximum) length of the chunk in bytes the function will read of the value and will store in the `value_data` buffer.
- `value_offset`: the offset of the chunk where the function should start storing the value in the provided buffer, i.e. the number of bytes the functions should skip from the retrieved value before storing the data in the `value_data` in number of bytes.
- `result`: The number of bytes the function writes in `value_data` if the value exists or $2^{32} - 1$ if the entry does not exist under the specified key.

A.1.9 To be Specced

- `ext_clear_child_storage`
- `ext_exists_child_storage`
- `ext_get_allocated_child_storage`
- `ext_get_child_storage_into`
- `ext_get_storage_into`
- `ext_kill_child_storage`
- `ext_set_child_storage`
- `ext_storage_changes_root`
- `ext_storage_root`

A.2 Memory

A.2.1 `ext_malloc`

Allocates memory of a requested size in the heap.

Prototype:
```
(func $ext_malloc
  (param $size i32) (result i32))
```

Arguments:

- `size:` the size of the buffer to be allocated in number of bytes.

Result:

a memory address pointing at the beginning of the allocated buffer.

A.2.2 `ext_free`

Deallocates a previously allocated memory.

Prototype:
```
(func $ext_free
      (param $addr i32))
```

Arguments:

- `addr`: a 32bit memory address pointing at the allocated memory.

A.2.3 Input/Output

- `ext_print_hex`
- `ext_print_num`
- `ext_print_utf8`

A.3 Cryptograhpic auxiliary functions

A.3.1 ext_blake2_256

Computes the Blake2s hash of a given byte array.

Prototype:

```
(func (export "ext_blake2_256")
      (param $data i32) (param  $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer and containing the byte array to be hashed.
- `len`: the length of the byte array in bytes.
- `out`: a memory address pointing at the beginning of a 32-byte byte array contanining the Blake2s hash of the data.

A.3.2 `ext_ed25519_verify`

Given a message signed by the ED25519 signature algorithm alongside with its signature and the allegedly signer public key, it verifies the validity of the signature by the provided public key.

Prototype:

```
(func $ext_ed25519_verify
      (param $msg_data i32) (param $msg_len i32) (param $sig_data i32)
      (param $pubkey_data i32) (result i32))
```

Arguments:

- `msg_data`: a pointer to the buffer containing the message body.
- `msg_len`: an `i32` integer indicating the size of the message buffer in bytes.
- `sig_data`: a pointer to the 64 byte memory buffer containing the ED25519 signature corresponding to the message.
- `pubkey_data`: a pointer to the 32 byte buffer containing the public key and corresponding to the secret key which has signed the message.

- `result`: an integer value equal to 0 indicating the validity of the signature or a nonzero value otherwise.

A.3.3 To be Specced

- `ext_twox_128`
- `ext_twox_256`

A.4 Sandboxing

A.4.1 To be Specced

- `ext_sandbox_instance_teardown`
- `ext_sandbox_instantiate`
- `ext_sandbox_invoke`
- `ext_sandbox_memory_get`
- `ext_sandbox_memory_new`
- `ext_sandbox_memory_set`
- `ext_sandbox_memory_teardown`

A.4.2 Misc

A.4.3 To be Specced

- `ext_chain_id`

A.5 Not implemented in Polkadot-JS

Bibliography

[Ali19] Alistair Stewart. GRANDPA: A Byzantine Finality Gadgets, 2019.

[DGKR18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[Gro19] W3F Research Group. Blind Assignment for Blockchain Extension. Technical Specification, Web 3.0 Foundation, http://research.web3.foundation/en/latest/polkadot/BABE/Babe/, 2019.