Polkadot Runtime Environment

Protocol Specification

January 27, 2020

Table of contents

Chapter 1

Background

1.1. Introduction

Formally, Polkadot is a replicated sharded state machine designed to resolve the scalability and interoperability among blockchains. In Polkadot vocabulary, shards are called *parachains* and Polkadot *relay chain* is part of the protocol ensuring global consensus among all the parachains. The Polkadot relay chain protocol, henceforward called *Polkadot protocol*, can itself be considered as a replicated state machine on its own. As such, the protocol can be specified by identifying the state machine and the replication strategy.

From a more technical point of view, the Polkadot protocol has been divided into two parts, the *Runtime* and the *Runtime environment* (RE). The Runtime comprises most of the state transition logic for the Polkadot protocol and is designed and expected to be upgradable as part of the state transition process. The Runtime environment consists of parts of the protocol, shared mostly among peer-to-peer decentralized cryptographically-secured transaction systems, i.e. blockchains whose consensus system is based on the proof-of-stake. The RE is planned to be stable and static for the lifetime duration of the Polkadot protocol.

With the current document, we aim to specify the RE part of the Polkadot protocol as a replicated state machine. After defining the basic terms in Chapter 1, we proceed to specify the representation of a valid state of the Protocol in Chapter 2. In Chapter 3, we identify the protocol states, by explaining the Polkadot state transition and discussing the detail based on which Polkadot RE interacts with the state transition function, i.e. Runtime. Following, we specify the input messages triggering the state transition and the system behaviour. In Chapter 5, we specify the consensus protocol, which is responsible for keeping all the replica in the same state. Finally, the initial state of the machine is identified and discussed in Appendix C. A Polkadot RE implementation which conforms with this part of the specification should successfully be able to sync its states with the Polkadot network.

1.2. Definitions and Conventions

Definition 1.1. *A **Discrete State Machine (DSM)** is a state transition system whose set of states and set of transitions are countable and admits a starting state. Formally, it is a tuple of*

$$(\Sigma, S, s_0, \delta)$$

where

- Σ *is the countable set of all possible transitions.*
- S *is a countable set of all possible states.*
- $s_0 \in S$ *is the initial state.*
- δ *is the state-transition function, known as **Runtime** in the Polkadot vocabulary, such that*

$$\delta : S \times \Sigma \to S$$

DEFINITION 1.2. *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 $(v_1, ..., v_n)$ 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 1.3. ***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 1.4. *By a **sequences of bytes** or a **byte array**, b, of length n, we refer to*

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

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

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

DEFINITION 1.6. *For a given byte b the **bitwise representation** of b is defined as*

$$b := b^7 ... b^0$$

where

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

DEFINITION 1.7. *By the **little-endian** representation of a non-negative integer, I, represented as*

$$I = (B_n ... B_0)_{256}$$

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

$$b_i := B_i$$

Accordingly, define the function Enc_{LE}:

$$\begin{array}{rccc} \text{Enc}_{\text{LE}}: & \mathbb{Z}^+ & \to & \mathbb{B} \\ & (B_n ... B_0)_{256} & \mapsto & (B_0, B_1, ..., B_n) \end{array}$$

DEFINITION 1.8. *By **`UINT32`** we refer to a non-negative integer stored in a byte array of length 4 using little-endian encoding format.*

Definition 1.9. *A **blockchain** C is a directed path graph. Each node of the graph is called **Block** and indicated by **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 $B_1 \to B_2$ we say B_2 is the **parent** of B_1 and we indicate it by*

$$B_2 := P(B_1)$$

Definition 1.10. *By **UNIX time**, we refer to the unsigned, little-endian encoded 64-bit integer which stores the number of **milliseconds** that have elapsed since the Unix epoch, that is the time 00:00:00 UTC on 1 January 1970, minus leap seconds. Leap seconds are ignored, and every day is treated as if it contained exactly 86400 seconds.*

1.2.1. 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 1.11. *The **block tree** of a blockchain, denoted by* BT *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.*

When a block in the block tree gets finalized, there is an opportunity to prune the block tree to free up resources into branches of blocks that do not contain all of the finalized blocks or those that can never be finalized in the blockchain. For a definition of finality, see Section 5.3.

Definition 1.12. *By **Pruned Block Tree**, denoted by* PBT*, we refer to a subtree of the block tree obtained by eliminating all branches which do not contain the most recent finalized blocks, as defined in Definition 5.27. By **pruning**, we refer to the procedure of* $\mathrm{BT} \leftarrow \mathrm{PBT}$*. When there is no risk of ambiguity and is safe to prune BT, we use* BT *to refer to* PBT*.*

Definition 1.13 gives the means to highlight various branches of the block tree.

Definition 1.13. *Let G be the root of the block tree and B be one of its nodes. By* **Chain(B)**, *we refer to the path graph from G to B in (P)*BT*. Conversely, for a chain $C=$*Chain(B)*, we define **the head of C** to be B, formally noted as $B :=$* Head(C)*. We define $|C|$, the length of C as a path graph. If B' is another node on* Chain(B)*, then by* SubChain(B', B) *we refer to the subgraph of* Chain(B) *path graph which contains both B and B'. Accordingly, $\mathbb{C}_{B'}$((P)*BT*) is the set of all subchains of* (P)BT *rooted at B'. The set of all chains of* (P)BT*, $\mathbb{C}_G$((P)*BT*) is denoted by $\mathbb{C}$((P)BT) or simply $\mathbb{C}$, for the sake of brevity.*

Definition 1.14. *We define the following complete order over $\mathbb{C}$ such that for $C_1, C_2 \in \mathbb{C}$ if $|C_1| \neq |C_2|$ we say $C_1 > C_2$ if and only if $|C_1| > |C_2|$.*

If $|C_1| = |C_2|$ we say $C_1 > C_2$ if and only if the block arrival time of Head(C_1) *is less than the block arrival time of* Head(C_2) *as defined in Definition 5.10. We define the* **Longest-Chain(BT)** *to be the maximum chain given by this order.*

Definition 1.15. Longest-Path(BT) *returns the path graph of* (P)BT *which is the longest among all paths in* (P)BT *and has the earliest block arrival time as defined in Definition 5.10.* Deepest-Leaf(BT) *returns the head of* Longest-Path(BT) *chain.*

Because every block in the blockchain contains a reference to its parent, it is easy to see that the block tree is de facto a tree. A block tree naturally imposes partial order relationships on the blocks as follows:

Definition 1.16. *We say **B is descendant of B'**, formally noted as **$B > B'$** if B is a descendant of B' in the block tree.*

Chapter 2

State Specification

2.1. State Storage and 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 assumption either on the size of the key nor on the size of the data stored under them, besides the fact that they are byte arrays with specific upper limits on their length. The limit is imposed by the encoding algorithms to store the key and the value in the storage trie.

2.1.1. Accessing System Storage

Polkadot RE implements various functions to facilitate access to the system storage for the Runtime. See Section 3.1 for a an explaination 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 2.1. *The **StoredValue** function retrieves the value stored under a specific key in the state storage and is formally defined as :*

$$\text{StoredValue:} \quad \mathcal{K} \to \mathcal{V}$$
$$k \mapsto \begin{cases} v & \text{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases}$$

where $\mathcal{K} \subset \mathbb{B}$ and $\mathcal{V} \subset \mathbb{B}$ are respectively the set of all keys and values stored in the state storage.

2.1.2. 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 3.6), 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 1.3. 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 2.2. *For the purpose of labeling the branches of the Trie, the key k is encoded to k_{enc} using KeyEncode functions:*

$$k_{\text{enc}} := (k_{\text{enc}_1}, ..., k_{\text{enc}_{2^n}}) := \text{KeyEncode}(k) \tag{2.1}$$

such that:

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

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*:*

$$(b_i^1, b_i^2) := (b_i / 16, b_i \bmod 16)$$

, 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.

2.1.3. Trie Structure

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

NOTATION 2.3. *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 2.4. *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 2.5. *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:*

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

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 2.6.

DEFINITION 2.6. *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 2.1 and a* ***partial key***, $\mathbf{pk}_N$ *of length $0 \leqslant l_{\text{pk}_N} \leqslant 65535$ in nibbles such that:*

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

where pk_N *is a suffix subsequence of* k_N*; i is the length of* $\mathrm{pk}_N^{\mathrm{Agr}}$ *in nibbles and so we have:*

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

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 2.7.

Definition 2.7. *For* $N \in \mathcal{N}_b$ *and* N_c *child of N, we define* **Index$_N$** *function as:*

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

such that

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

Assuming that P_N is the path (see Definition 1.2) from the Trie root to node N, Algorithm 2.1 rigorously demonstrates how to build $\mathrm{pk}_N^{\mathrm{Agr}}$ while traversing P_N.

Algorithm 2.1

Aggregate-Key($P_N := (\mathrm{TrieRoot} = N_1, ..., N_j = N)$)

Definition 2.8. *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 2.9,2.6, B.9 and 2.11, respectively.*

Definition 2.9. *The* ***node header*** *of node N,* Head_N*, consists of* $l + 1 \geqslant 1$ *bytes* $\mathrm{Head}_{N,1}, ..., \mathrm{Head}_{N,l+1}$ *such that:*

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:*

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

where $\mathcal{K}$ *is defined in Definition 2.1.*

$\mathrm{Head}_{N,1}^{0-5}$*, 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 $\|\mathbf{pk_N}\|_{\mathbf{nib}}$ *is the length of* pk_N *in number nibbles.* $\mathrm{Head}_{N,2}, ..., \mathrm{Head}_{N,l+1}$ *bytes are determined by Algorithm 2.2.*

Algorithm 2.2

PARTIAL-KEY-LENGTH-ENCODING($\mathrm{Head}_{N,1}^{6-7}, \mathrm{pk}_N$)

2.1.4. 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 2.10 to encode and decode information stored in a branch node.

DEFINITION 2.10. *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:*

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

where

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

DEFINITION 2.11. *For a given node* N*, the* ***subvalue*** *of* N*, formally referred to as* sv_N*, is determined as follows: in a case which:*

$$\mathrm{sv}_N := \begin{cases} \mathrm{Enc_{SC}}(\mathrm{StoredValue}(k_N)) & N \textit{ is a leaf node} \\ \mathrm{ChildrenBitmap}(N) \| \mathrm{Enc_{SC}}(H(N_{C_1})) \, ... \, \mathrm{Enc_{SC}}(H(N_{C_n})) || \mathrm{Enc_{SC}}(\mathrm{StoredValue}(k_N)) & N \textit{ is a branch node} \end{cases}$$

Where $N_{C_1} ... N_{C_n}$ with $n \leqslant 16$ are the children nodes of the branch node N and $\mathrm{Enc_{SC}}$, StoredValue, H, and ChildrenBitmap(N) are defined in Definitions B.1,2.1, 2.12 and 2.10 respectively.

The Trie deviates from a traditional Merkle tree where node value, v_N (see Definition 2.8) is presented instead of its hash if it occupies less space than its hash.

DEFINITION 2.12. *For a given node* N*, the* ***Merkle value*** *of* N*, denoted by* $H(N)$ *is defined as follows:*

$$\begin{aligned} & H : \mathbb{B} \to \cup_{i \to 0}^{32} \mathbb{B}_{32} \\ & H(N) : \begin{cases} v_N & \|v_N\| < 32 \ \textit{ and } \ N \neq R \\ \mathrm{Blake2}b(v_N) & \|v_N\| \geqslant 32 \ \textit{ or } \ N = R \end{cases} \end{aligned}$$

Where v_N *is the node value of* N *defined in Definition 2.8 and* R *is the root of the Trie. The* ***Merkle hash*** *of the Trie is defined to be* $H(R)$*.*

Chapter 3

State Transition

Like any transaction-based transition system, Polkadot state changes via executing an ordered set of instructions. These instructions are known as *extrinsics*. In Polkadot, the execution logic of the state-transition function is encapsulated in Runtime as defined in Definition 1.1. Runtime is presented as a Wasm blob in order to be easily upgradable. Nonetheless, the Polkadot Runtime Environment needs to be in constant interaction with Runtime. The detail of such interaction is further described in Section 3.1.

In Section 3.2, we specify the procedure of the process where the extrinsics are submitted, pre-processed and validated by Runtime and queued to be applied to the current state.

Polkadot, as with most prominent distributed ledger systems that make state replication feasible, journals and batches a series of extrinsics together in a structure known as a *block* before propagating to the other nodes. The specification of the Polkadot block as well as the process of verifying its validity are both explained in Section 3.3.

3.1. Interactions with Runtime

Runtime as defined in Definition • 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. The general procedure to interact with Runtime is described in Algorithm 3.1.

Algorithm 3.1

Interact-With-Runtime(F: the runtime entry,
$H_b(B)$: Block hash indicating the state at the end of B,
$A_1, A_2, ..., A_n$: arguments to be passed to the runtime entry)

In this section, we describe the details upon which the Polkadot RE is interacting with the Runtime. In particular, Set-State-At and Call-Runtime-Entry procedures called in Algorithm 3.1 are explained in Notation 3.2 and Definition 3.10 respectively. R_B is the Runtime code loaded from $\mathcal{S}_B$, as described in Notation 3.1, and $\mathcal{RE}_B$ is the Polkadot RE API, as described in Notation E.1.

3.1.1. 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:

$$b := 3A,63,6F,64,65$$

which is the byte array of ASCII representation of string ":code" (see Section C). For any call to the Runtime, Polkadot RE makes sure that it has the Runtime corresponding to the state in which the entry has been called. This is, in part, because the calls to Runtime have potentially the ability to change the Runtime code and hence Runtime code is state sensitive. Accordingly, we introduce the following notation to refer to the Runtime code at a specific state:

Notation 3.1. *By R_B, we refer to the Runtime code stored in the state storage whose state is set at the end of the execution of block B.*

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

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

3.1.2. 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***.

Definition 3.2 introduces the notation for calling the runtime entry which is used whenever an algorithm of Polkadot RE needs to access the runtime.

Notation 3.2. *By*

$$\textsc{Call-Runtime-Entry}(R, \mathcal{RE}, \texttt{Runtime-Entry}, A, A_{\text{len}})$$

we refer to the task using the executor to invoke the `Runtime-Entry` *while passing an $A_1, ..., A_n$ argument to it and using the encoding described in Section 3.1.2.2.*

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

3.1.2.1. 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 F. This can be done for example by loading another Wasm module alongside the runtime which imports these functions from Polkadot RE as host functions.

3.1.2.2. Sending Arguments to Runtime

In general, all data exchanged between Polkadot RE and the Runtime is encoded using SCALE codec described in Section B.1. As a Wasm function, all runtime entries have the following identical signatures:

```
(func $runtime_entry (param $data i32) (param $len i32) (result i64))
```

In each invocation of a Runtime entry, the argument(s) which are supposed to be sent to the entry, need to be encoded using SCALE codec into a byte array B using the procedure defined in Definition B.1.

The Executor then needs to retrieve the Wam memory buffer of the Runtime Wasm module and extend it to fit the size of the byte array. Afterwards, it needs to copy the byte array B value in the correct offset of the extended buffer. Finally, when the Wasm method `runtime_entry`, corresponding to the entry is invoked, two UINT32 integers are sent to the method as arguments. The first argument `data` is set to the offset where the byte array B is stored in the Wasm the extended shared memory buffer. The second argument `len` sets the length of the data stored in B., and the second one is the size of B.

3.1.2.3. The Return Value from a Runtime Entry

The value which is returned from the invocation is an `i64` integer, representing two consecutive `i32` integers in which the least significant one indicates the pointer to the offset of the result returned by the entry encoded in SCALE codec in the memory buffer. The most significant one provides the size of the blob.

3.2. Extrinsics

The block body consists of an array of extrinsics. In a broad sense, extrinsics are data from outside of the state which can trigger the state transition. This section describes the specifications of the extrinsics and their inclusion in the blocks.

3.2.1. Preliminaries

The extrinsics are divided in two main categories and defined as follows:

Definition 3.3. ***Transaction extrinsics*** *are extrinsics which are signed using either of the key types described in section A.5 and broadcasted between the nodes.* ***Inherents extrinsics*** *are unsigned extrinsics which are generated by Polkadot RE and only included in the blocks produced by the node itself. They are broadcasted as part of the produced blocks rather than being gossiped as individual extrinsics.*

Polkadot RE does not specify or limit the internals of each extrinsics and those are dealt with by the Runtime. From Polkadot RE point of view, each extrinsics is simply a SCALE-encoded blob (see Section B.1).

3.2.2. Transactions

3.2.2.1. Transaction Submission

Transaction submission is made by sending a *Transactions* network message. The structure of this message is specified in Section D.1.5. Upon receiving a Transactions message, Polkadot RE decodes and decouples the transactions and calls `validate_trasaction` Runtime entry, defined in Section F.2.7, to check the validity of each received transaction. If `validate_transaction` considers the submitted transaction as a valid one, Polkadot RE makes the transaction available for the consensus engine for inclusion in future blocks.

3.2.3. Transaction Queue

A Block producer node should listen to all transaction messages. This is because the transactions are submitted to the node through the *transactions* network message specified in Section D.1.5. Upon receiving a transactions message, Polkadot RE separates the submitted transactions in the transactions message into individual transactions and passes them to the Runtime by executing Algorithm 3.2 to validate and store them for inclusion into future blocks. To that aim, Polkodot RE should keep a *transaction pool* and a *transaction queue* defined as follows:

Definition 3.4. *The* ***Transaction Queue*** *of a block producer node, formally referred to as* TQ *is a data structure which stores the transactions ready to be included in a block sorted according to their priorities (Definition D.1.5). The* ***Transaction Pool****, formally referred to as* TP*, is a hash table in which Polkadot RE keeps the list of all valid transactions not in the transaction queue.*

Algorithm 3.2 updates the transaction pool and the transaction queue according to the received message:

Algorithm 3.2

Validate-Transactions-and-Store(M_T: Transaction Message)

In which

- Longest-Chain is defined in Definition 1.14.
- `TaggedTransactionQueue_validate_transaction` is a Runtime entry specified in Section F.2.7 and Requires(R), Priority(R) and Propagate(R) refer to the corresponding fields in the tuple returned by the entry when it deems that T is valid.
- Provided-Tags(T) is the list of tags that transaction T provides. Polkadot RE needs to keep track of tags that transaction T provides as well as requires after validating it.
- Insert-At(TQ, T, Requires(R), Priority(R)) places T into TQ approperietly such that the transactions providing the tags which T requires or have higher priority than T are ahead of T.
- Maintain-Transaction-Pool is described in Algorithm 3.3.
- Propagate(T) include T in the next *transactions message* sent to all peers of Polkadot RE node.

Algorithm 3.3

Maintain-Transaction-Pool

3.2.3.1. Inherents

Block inherent data represents the totality of inherent extrinsics included in each block. This data is collected or generated by the Polkadot RE and handed to the Runtime for inclusion in the block. It's the responsability of the RE implementation to keep track of those values. Table 3.1 lists these inherent data, identifiers, and types. [define uncles]

Identifier	Value type	Description
timstap0	u64	Unix epoch time in number of milliseconds
babeslot	u64	Babe Slot Number[5.5] of the current building block
finalnum	compact integer[B.8]	Header number[3.6] of the last finalized block
uncles00	array of block headers	Provides a list of potential uncle block headers[3.6] for a given block

Table 3.1. List of inherent data

Definition 3.5. Inherent-Data *is a hashtable (Definition B.5) representing the totality of inherent extrinsics included in each block. The entries of this hash table which are listed in Table 3.1 are collected or generated by the Polkadot RE and then handed to the Runtime for inclusion as dercribed in Algorithm 5.7. The identifiers are 8-byte values.*

3.3. State Replication

Polkadot nodes replicate each other's state by syncing the history of the extrinsics. This, however, is only practical if a large set of transactions are batched and synced at the time. The structure in which the transactions are journaled and propagated is known as a block (of extrinsics) which is specified in Section 3.3.1. Like any other replicated state machines, state inconsistency happens across Polkadot replicas. Section 3.3.4 is giving an overview of how a Polkadot RE node manages multiple variants of the state.

3.3.1. Block Format

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.

3.3.1.1. 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 3.6. *The* ***header of block*** B, $\mathbf{Head}(B)$ *is a 5-tuple containing the following elements:*

- ***parent_hash:*** *is the 32-byte Blake2b hash (see Section A.2) 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 state 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. The* extrinsics_root *is set by the runtime and its value is opaque to Polkadot RE. 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 as well as consensus-related data including the block signature. This field is indicated as* $\boldsymbol{H_d}$ *and its detailed format is defined in Definition 3.7*

Definition 3.7. *The header* ***digest*** *of block B formally referred to by* $\boldsymbol{H_d(B)}$ *is an array of* ***digest items*** H_d^i*'s , known as digest items of varying data type (see Definition B.3) such that*

$$H_d(B) := H_d^1, ..., H_d^n$$

where each digest item can hold one of the type described in Table 3.2:

Type Id	*Type name*	*sub-components*
2	*Changes trie root*	$\mathbb{B}_{32}$
6	*Pre-Runtime*	$E_{\text{id}}, \mathbb{B}$
4	*Consensus Message*	$E_{\text{id}}, \mathbb{B}$
5	*Seal*	$E_{\text{id}}, \mathbb{B}$

Table 3.2. *The detail of the varying type that a digest item can hold.*

Where E_{id} *is the unique consensus engine identifier defined in Section D.1.6. and*

- ***Changes trie root*** *contains the root of changes trie at block B.*
- ***Pre-runtime*** *digest item represents messages produced by a consensus engine to the Runtime.*
- ***Consensus Message*** *digest item represents a message from the Runtime to the consensus engine (see Section 5.1.2).*

- ***Seal*** *is the data produced by the consensus engine and proving the authorship of the block producer. In particular, the Seal digest item must be the last item in the digest array and must be stripped off before the block is submitted to the Runtime for validation and be added back to the digest afterward. The detail of the Seal digest item is laid out in Definition 5.13.*

Definition 3.8. *The* ***Block Header Hash of Block B***, $\boldsymbol{H_h(B)}$, *is the hash of the header of block B encoded by simple codec:"*

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

3.3.1.2. 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 3.3.1.1 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.

3.3.1.3. Block Body

The Block Body consists of array extrinsics each encoded as a byte array. The internal of extrinsics is completely opaque to Polkadot RE. As such, it forms the point of Polkadot RE, and is simply a SCALE encoded array of byte arrays. Formally:

Definition 3.9. *The* ***body of Block*** *B represented as* **Body($\boldsymbol{B}$)** *is defined to be*

$$\text{Body}(B) := \text{Enc}_{\text{SC}}(E_1, ..., E_n)$$

Where each $E_i \in \mathbb{B}$ *is a SCALE encoded extrinsic.*

3.3.2. Block Submission

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 for example by means of Block response network message (see Section D.1.3) and from the consensus engine.

3.3.3. Block Validation

Both the Runtime and the Polkadot RE need to work together to assure block validity. A block is deemed valid if the block author had the authorship right for the slot during which the slot was built as well as if the transactions in the block constitute a valid transition of states. The former criterion is validated by Polkadot RE according to the block production consensus protocol. The latter can be verified 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 3.4

Import-and-Validate-Block(B, Just(B))

For the definition of the finality and the finalized block see Section 5.3. PBT is the pruned block tree defined in Definition 1.11. Verify-Authorship-Right is part of the block production consensus protocol and is described in Algorithm 5.5.

3.3.4. Managaing Multiple Variants of State

Unless a node is committed to only update its state according to the finalized block (See Definition 5.27), it is inevitable for the node to store multiple variants of the state (one for each block). This is, for example, necessary for nodes participating in the block production and finalization.

While the state trie structure described in Section 2.1.3 facilitates and optimizes storing and switching between multiple variants of the state storage, Polkadot RE does not specify how a node is required to accomplish this task. Instead, Polkadot RE is required to implement Set-State-At operation which behaves as defined in Definition 3.10:

Definition 3.10. *The function*

Set-State-At(B)

in which B is a block in the block tree (See Definition 1.11), sets the content of state storage equal to the resulting state of executing all extrinsics contained in the branch of the block tree from genesis till block B including those recorded in Block B.

For the definition of the state storage see Section 2.1.

Chapter 4

Network Protocol

Warning 4.1. Polkadot network protocol is work-in-progress. The API specification and usage may change in future.

This chapter offers a high-level description of the network protocol based on [Tec19]. Polkadot network protocol relies on *libp2p*. Specifically, the following libp2p modules are being used in the Polkadot Networking protocol:

- mplex.
- yamux
- secio
- noise
- kad (kademlia)
- identity
- ping

For more detailed specification of these modules and the Peer-to-Peer layer see libp2p specification document [lab19].

4.1. Node Identities and Addresses

Similar to other decentralized networks, each Polkadot RE node possesses a network private key and a network public key representing an ED25519 key pair [LJ17].

[SPEC: local node's keypair must be passed as part of the network configuration.]

Definition 4.2. ***Peer Identity***, *formally noted by* P_{id} *is derived from the node's public key as follows:* *[SPEC: How to derive* P_{id}*]* *and uniquely identifies a node on the network.*

Because the P_{id} is derived from the node's public key, running two or more instances of Polkadot network using the same network key is contrary to the Polkadot protocol.

All network communications between nodes on the network use encryption derived from both sides' keys.

[SPEC: p2p key derivation]

4.2. Discovery Mechanisms

In order for a Polkadot node to join a peer-to-peer network, it has to know a list of Polkadot nodes that already take part in the network. This process of building such a list is referred to as *Discovery*. Each element of this list is a pair consisting of the peer's node identities and their addresses.

[SPEC: Node address]

Polkadot discovery is done through the following mechanisms:

- *Bootstrap nodes*: These are hard-coded node identities and addresses passed alongside with the network configuration.
- *mDNS*, performing a UDP broadcast on the local network. Nodes that listen may respond with their identity as described in the mDNS section of [lab19]. (Note: mDNS can be disabled in the network configuration.)
- *Kademlia random walk*. Once connected to a peer node, a Polkadot node can perform a random Kademlia 'FIND_NODE' requests for the nodes [which nodes?] to respond by propagating their view of the network.

4.3. TRANSPORT PROTOCOL

A Polkadot node can establish a connection with nodes in its peer list. All the connections must always use encryption and multiplexing. While some nodes' addresses (eg. addresses using '/quic') already imply the encryption and/or multiplexing to use, for others the "multistream-select" protocol is used in order to negotiate an encryption layer and/or a multiplexing layer.

The following transport protocol is supported by a Polkadot node:

- *TCP/IP* for addresses of the form '/ip4/1.2.3.4/tcp/5'. Once the TCP connection is open, an encryption and a multiplexing layers are negotiated on top.
- *WebSockets* for addresses of the form '/ip4/1.2.3.4/tcp/5/ws'. A TC/IP connection is open and the WebSockets protocol is negotiated on top. Communications then happen inside WebSockets data frames. Encryption and multiplexing are additionally negotiated again inside this channel.
- DNS for addresses of the form '/dns4/example.com/tcp/5' or '/dns4/example.com/tcp/5/ws'. A node's address can contain a domain name.

4.3.1. Encryption

The following encryption protocols from libp2p are supported by Polkadot protocol:

* **Secio**: A TLS-1.2-like protocol but without certificates [lab19]. Support for secio will likely to be deprecated in the future.
* **Noise**: Noise is a framework for crypto protocols based on the Diffie-Hellman key agreement [Per18]. Support for noise is experimental and details may change in the future.

4.3.2. Multiplexing

The following multiplexing protocols are supported:

- **Mplex**: Support for mplex will be deprecated in the future.
- **Yamux**.

4.4. Substreams

Once a connection has been established between two nodes and is able to use multiplexing, substreams can be opened. When a substream is open, the *multistream-select* protocol is used to negotiate which protocol to use on that given substream.

4.4.1. Periodic Ephemeral Substreams

A Polkadot RE node should open several substreams. In particular, it should periodically open ephemeral substreams in order to:

- ping the remote peer and check whether the connection is still alive. Failure for the remote peer to reply leads to a disconnection. This uses the libp2p *ping* protocol specified in [lab19].
- ask information from the remote. This is the *identity* protocol specified in [lab19].
- send Kademlia random walk queries. Each Kademlia query is done in a new separate substreams. This uses the libp2p *Kademlia* protocol specified in [lab19].

4.4.2. Polkadot Communication Substream

For the purposes of communicating Polkadot messages, the dailer of the connection opens a unique substream. Optionally, the node can keep a unique substream alive for this purpose. The name of the protocol negotiated is based on the *protocol ID* passed as part of the network configuration. This protocol ID should be unique for each chain and prevents nodes from different chains to connect to each other.

The structure of SCALE encoded messages sent over the unique Polkadot communication substream is described in Appendix D.

Once the substream is open, the first step is an exchange of a *status* message from both sides described in Section D.1.1.

Communications within this substream include:

- Syncing. Blocks are announced and requested from other nodes.
- Gossiping. Used by various subprotocols such as GRANDPA.
- Polkadot Network Specialization: [spec this protocol for polkadot].

Chapter 5

Consensus

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.

5.1. Common Consensus Structures

5.1.1. Consensus Authority Set

Because Polkadot is a proof-of-stake protocol, each of its consensus engine has its own set of nodes, represented by known public keys which have the authority to influence the protocol in pre-defined ways explained in this section. In order to verifiy the validity of each block, Polkadot node must track the current list of authorities for that block as formalised in Definition 5.1

Definition 5.1. *The **authority list** of block B for consensus engine C noted as $\mathbf{Auth_C(B)}$ is an array of pairs of type:*

$$(\mathrm{Pk}_A, W_A)$$

P_A is the session public key of authority A as defined in Definition A.4. And W_A is a `u64` value, indicating the authority weight which is set to equal to 1. The value of $\mathrm{Auth}_C(B)$ is part of the Polkadot state. The value for $\mathrm{Auth}_C(B_0)$ is set in the genesis state (see Section C) and can be retrieved using a runtime entery corresponding to consensus engine C.

Note that in Polkadot, all authorities have the weight equal to 1. The weight W_A in Definition 5.1 exists for potential improvements in the protocol and could have a use-case in the future.

5.1.2. Runtime-to-Consensus Engine Message

The authority lists (see Definition 5.1) is part of Polkadot state and the Runtime has the authority of updating the lists in the course of state transitions. The runtime inform the corresponding consensus engine about the changes in the authority set by means of setting a consensus message digest item as defined in Definition 5.2, in the block header of block B during which course of exectution the transition in the authority set has occured.

Definition 5.2. *Consensus Message is digest item of type 4 as defined in Definition 3.7 and consists of the pair:*

$$(E_{\mathrm{id}}, \mathrm{CM})$$

Where $E_{\mathrm{id}} \in \mathbb{B}_4$ is the consensus engine unique identifier which can hold the following possible values

$$E_{\mathrm{id}} := \begin{cases} \text{"BABE"} & \text{For messages related to BABE protocol refered to as } E_{\mathrm{id}}(\mathrm{BABE}) \\ \text{"FRNK"} & \text{For messages related to GRANDPA protocol referred to as } E_{\mathrm{id}}(\mathrm{FRNK}) \end{cases}$$

and CM is of varying data type which can hold one of the type described in Table 5.1:

Type Id	***Type***	***Sub-components***
1	*Scheduled Change*	(Auth_C, N_B)
2	*[RESERVED]*	*[RESERVED]*
3	*On Disabled*	Auth_{ID}
4	*Pause*	N_B
5	*Resume*	N_B

Table 5.1. *The consensus digest item for GRANDPA authorities*

Where Auth_C is defined in Definition 5.1, N_B is the number of blocks to delay the change. Auth_{ID} *is a 64 bit integer pointing to the authority list of the current block. Type Id 2 is reserved for future development [Spec Force Change].*

Polkadot RE should inspect the digest header of each block and delegates consesus messages to their consensus engines. Consensus engine should react based on the type of consensus messages they receives as follows:

- **Scheduled Change**: Schedule an authority set change. The earliest digest of this type in a single block will be respected. No change should be scheduled if one is already and the delay has not passed completely. If such an inconsitency occures, the scheduled change should be ignored.
- **On Disabled**: The authority set index with given index is disabled until the next change.
- **Pause**: A signal to pause the current authority set after the given delay. After finalizing the block at delay the authorities should stop voting.
- **Resume**: A signal to resume the current authority set after the given delay. After authoring the block at delay the authorities should resume voting.

The active GRANDPA authorities can only vote for blocks that occured after the finalized block in which they were selected. Any votes for blocks before the `Scheduled Change` came into effect get rejected.

5.2. Block Production

Polkadot RE uses BABE protocol [Gro19] for block production. It is 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 5.1 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 by eliminating branches which do not include the most recent finalized blocks according to Definition 1.12.

5.2.1. Preliminaries

DEFINITION 5.3. *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 5.4. ***Block authoring session key pair*** $(\mathbf{sk}_j^s, \mathbf{pk}_j^s)$ *is an SR25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (see Definition A.1) and is used to sign the produced block as well as to compute its lottery values in Algorithm 5.1.*

Definition 5.5. *A block production* ***epoch****, formally referred 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, and 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* ***slots****, sequentially indexed in each epoch. The index of each slot is called* ***slot number****. The equal length duration of each slot is called the* ***slot duration*** *and indicated by $\mathcal{T}$. Each slot is awarded to a subset of block producers during which they are allowed to generate a block.*

Notation 5.6. *We refer to the number of slots in epoch $\mathcal{E}_n$ by sc_n. sc_n is set to the* `duration` *field in the returned data from the call of the Runtime entry* `BabeApi_epoch` *(see F.2.5) at the beginning of each epoch. For a given block B, we use the notation $\boldsymbol{s_B}$ to refer to the slot during which B has been produced. Conversely, for slot s, $\mathcal{B}_s$ is the set of Blocks generated at slot s.*

Definition 5.7 provides an iterator over the blocks produced during an specific epoch.

Definition 5.7. *By* SubChain($\mathcal{E}_n$) *for epoch $\mathcal{E}_n$, we refer to the path graph of* BT *which contains all the blocks generated during the slots of epoch $\mathcal{E}_n$. When there is more than one block generated at a slot, we choose the one which is also on* Longest-Chain(BT)*.*

5.2.2. Block Production Lottery

Definition 5.8. ***Winning threshold*** *denoted by $\boldsymbol{\tau_{\varepsilon_n}}$ is the threshold which is used alongside with the result of Algorirthm 5.1 to decide if a block producer is the winner of a specific slot. τ_{ε_n} is calculated as follows:*

$$\tau_{\varepsilon_n} := 1 - (1-c)^{\frac{1}{|\text{AuthorityDirectory}^{\mathcal{E}_n}|}}$$

where AuthorityDirectory$^{\mathcal{E}_n}$ *is the set of BABE authorities for epoch ε_n and $c = \frac{c_{\text{nominator}}}{c_{\text{denominator}}}$. The pair ($c_{\text{nominator}}$, $c_{\text{denominator}}$) can be retrieve part of the data returned by a call into runtime entry* `BabeApi_configuration`*.*

A block producer aiming to produce a block during $\mathcal{E}_n$ should run Algorithm 5.1 to identify the slots it is awarded. These are the slots during which the block producer is allowed to build a block. The sk is the block producer lottery secret key and n is the index of epoch for whose slots the block producer is running the lottery.

Algorithm 5.1

Block-production-lottery(sk: session secret key of the producer,
n: epoch index)

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 Algorithm 5.4 and Section A.4 respectively.

5.2.3. Slot Number Calculation

It is essential for a block producer to calculate and validate the slot number at a certain point in time. Slots are dividing the time continuum in an overlapping interval. At a given time, the block producer should be able to determine the set of slots which can be associated to a valid block generated at that time. We formalize the notion of validity in the following definitions:

Definition 5.9. *The* ***slot tail****, formally referred to by* SlTl *represents the number of on-chain blocks that are used to estimate the slot time of a given slot. This number is set to be 1200.*

Algorithm 5.2 determines the slot time for a future slot based on the *block arrival time* associated with blocks in the slot tail defined in Definition 5.10.

DEFINITION 5.10. *The* ***block arrival time*** *of block B for node j formally represented by $\boldsymbol{T_B^j}$ is the local time of node j when node j has received the block B for the first time. If the node j itself is the producer of B, T_B^j is set equal to the time that the block is produced. The index j in T_B^j notation may be dropped and B's arrival time is referred to by T_B when there is no ambiguity about the underlying node.*

In addition to the arrival time of block B, the block producer also needs to know how many slots have passed since the arrival of B. This value is formalized in Definition 5.11.

DEFINITION 5.11. *Let s_i and s_j be two slots belonging to epochs $\mathcal{E}_k$ and $\mathcal{E}_l$. By* SLOT-OFFSET(s_i, s_j) *we refer to the function whose value is equal to the number of slots between s_i and s_j (counting s_j) on time continuum. As such, we have* SLOT-OFFSET$(s_i, s_i) = 0$.

Algorithm 5.2

SLOT-TIME(s: the slot number of the slot whose time needs to be determined)

$\mathcal{T}$ is the slot duration defined in Definition 5.5.

5.2.4. Block Production

At each epoch, each block producer should run Algorithm 5.3 to produce blocks during the slots it has been awarded during that epoch. The produced block needs to carry *BABE header* as well as the *block signature* as Pre-Runtime and Seal digest items defined in Definition 5.12 and 5.13 respectively.

DEFINITION 5.12. *The* ***BABE Header*** *of block B, referred to formally by $\boldsymbol{H_{\mathrm{BABE}}(B)}$ is a tuple that consists of the following components:*

$$(d, \pi, j, s)$$

in which:

- π, d: *are the results of the block lottery for slot s.*
- j: *is index of the block producer producing block in the current authority directory of current epoch.*
- s: *is the slot at which the block is produced.*

$H_{\mathrm{BABE}}(B)$ must be included as a diegst item of Pre-Runtime type in the header digest $H_d(B)$ as defined in Definition 3.7.

DEFINITION 5.13. *The* ***Block Signature*** *noted by S_B is computed as*

$$\mathrm{Sig}_{\mathrm{SR25519},\mathrm{sk}_j^s}(H_h(B))$$

S_B should be included in $H_d(B)$ as the Seal digest item according to Definition 3.7 of value:

$$(E_{\mathrm{id}}(\mathrm{BABE}), S_B)$$

in which, E_{id}(BABE) is the BABE consensus engine unique identifier defined in Section D.1.6. The Seal digest item is referred to as ***BABE Seal****.*

Algorithm 5.3

Invoke-Block-Authoring(sk, pk, n, BT: Current Block Tree)

Add-Digest-Item appends a digest item to the end of the header digest $H_d(B)$ according to Definition 3.7.

5.2.5. Epoch Randomness

At the end of epoch $\mathcal{E}_n$, each block producer is able to compute the randomness seed it needs in order to participate in the block production lottery in epoch $\mathcal{E}_{n+2}$. For epoch 0 and 1, the randomness seed is provided in the genesis state. The computation of the seed is described in Algorithm 5.4 which uses the concept of epoch subchain described in Definition 5.7.

Algorithm 5.4

Epoch-Randomness($n > 2$: epoch index)

In which value d_B is the VRF output computed for slot s_B by running Algorithm 5.1.

5.2.6. Verifying Authorship Right

When a Polkadot node receives a produced block, it needs to verify if the block producer was entitled to produce the block in the given slot by running Algorithm 5.5 where:

- T_B is B's arrival time defined in Definition 5.10.
- $H_d(B)$ is the digest sub-component of Head(B) defined in Definitions 3.6 and 3.7.
- The Seal D_s is the last element in the digest array $H_d(B)$ as defined in Definition 3.7.
- Seal-Id is the type index showing that a digest item of variable type is of *Seal* type (See Definitions B.4 and 3.7)
- AuthorityDirectory$^{\mathcal{E}_c}$ is the set of Authority ID for block producers of epoch $\mathcal{E}_c$.
- verify-Slot-Winner is defined in Algorithm 5.6.

Algorithm 5.5

Verify-Authorship-Right(Head$_s(B)$: The header of the block being verified)

Algorithm 5.6 is run as a part of the verification process, when a node is importing a block, in which:

- Epoch-Randomness is defined in Algorithm 5.4.
- $H_{\text{BABE}}(B)$ is the BABE header defined in Definition 5.12.
- Verify-VRF is described in Section A.4.
- τ is the winning threshold defined in 5.8.

Algorithm 5.6

VERIFY-SLOT-WINNER(B: the block whose winning status to be verified)

(d_B, π_B): Block Lottery Result for Block B,
s_n: the slot number,
n: Epoch index
AuthorID: The public session key of the block producer

5.2.7. Block Building Process

The blocks building process is triggered by Algorithm 5.3 of the consensus engine which runs Alogrithm 5.7.

Algorithm 5.7

BUILD-BLOCK(C_{Best}: The chain where at its head, the block to be constructed,
s: Slot number)

- Head(B) is defined in Definition 3.6.
- CALL-RUNTIME-ENTRY is defined in Notation 3.2.
- INHERENT-DATA is defined in Definition 3.5.
- TRANSACTION-QUEUE is defined in Definition 3.4.
- BLOCK-IS-FULL indicates that the maximum block size as been used.
- END-OF-SLOT indicates the end of the BABE slot as defined in Algorithm 5.2 respectively Definition 5.5.
- OK-RESULT indicates whether the result of `BlockBuilder_apply_extrinsics` is successfull. The error type of the Runtime function is defined in Definition [define error type].
- READY-EXTRINSICS-QUEUE indicates picking an extrinsics from the extrinsics queue (Definition 3.4).
- DROP indicates removing the extrinsic from the transaction queue (Definition 3.4).

5.3. FINALITY

Polkadot RE uses GRANDPA Finality protocol [Ste19] 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.

5.3.1. Preliminaries

DEFINITION 5.14. *A **GRANDPA Voter**, v, is represented by a key pair $(k_v^{\text{pr}}, v_{\text{id}})$ 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 have [change function name, only call at genesis, adjust V_B over the sections]*

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

where `grandpa_authorities` *is the entry into runtime described in Section F.2.6.*

Definition 5.15. ***GRANDPA state***, GS, *is defined as* *[verify V_ id and id_ V usage, unify]*

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

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 5.16. *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 3.6 and 3.8 respectively.*

Definition 5.17. *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 $\boldsymbol{V_v^{r,\mathbf{pv}}}$ *and* $\boldsymbol{V_v^{r,\mathbf{pc}}}$ *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 5.9. After defining what constitues a vote in GRANDPA, we define how GRANDPA counts votes.

Definition 5.18. *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:*

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

Definition 5.19. *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 5.20. *For validator* v*,* ***the set of observed direct votes for Block B in round r****, formally denoted by* $\mathrm{VD}^{r,\mathrm{stage}}_{\mathrm{obs}(v)}(B)$ *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 5.21. *We refer to* ***the set of total votes observed by voter v in sub-round "stage" of round r*** *by* $\boldsymbol{V^{r,\text{stage}}_{\text{obs}(v)}}$.

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

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

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:*

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

DEFINITION 5.22. *The current* ***pre-voted*** *block* $B^{r,\text{pv}}_v$ *is the block with*

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

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

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

DEFINITION 5.23. *We say that round r is* ***completable*** *if* $|V^{r,\text{pc}}_{\text{obs}(v)}| + \mathcal{E}^{r,\text{pc}}_{\text{obs}(v)} > \frac{2}{3}\mathbb{V}$ *and for all* $B' > B^{r,\text{pv}}_v$*:*

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

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

5.3.2. 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 5.9 for more details). These messages are specified in this section.

DEFINITION 5.24. *A vote casted by voter v should be broadcasted as a* ***message*** $\boldsymbol{M^{r,\text{stage}}_v}$ *to the network by voter v with the following structure:*

$$M^{r,\text{stage}}_v := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}}, \text{Enc}_{\text{SC}}(\text{stage}, V^{r,\text{stage}}_v, \text{Sig}_{\text{ED25519}}(\text{Enc}_{\text{SC}}(\text{stage}, V^{r,\text{stage}}_v, r, V_{\text{id}}), v_{\text{id}})$$

Where:

r:	*round number*	*64 bit integer*
V_{id}:	*incremental change tracker counter*	*64 bit integer*
v_{id}:	*Ed25519 public key of v*	*32 byte array*
stage:	*0 if it is the pre-vote sub-round*	*1 byte*
	1 if it the pre-commit sub-round	

Definition 5.25. *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

$$B' \geqslant B$$

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.

*We say $J^r(B)$ **justifies the finalization** of B if the number of valid signatures in $J^r(B)$ is greater than $\frac{2}{3}|\mathbb{V}_B|$.*

Definition 5.26. **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 5.25.

5.3.3. 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 5.8, whenever the membership of GRANDPA voters changes, r is set to 0 and V_{id} needs to be incremented.

Algorithm 5.8

Join-Leave-Grandpa-Voters ($\mathcal{V}$)

5.3.4. Voting Process in Round *r*

For each round r, an honest voter v must participate in the voting process by following Algorithm 5.9.

Algorithm 5.9

Play-Grandpa-round(r)

The condition of *completablitiy* is defined in Definition 5.23. Best-Final-Candidate function is explained in Algorithm 5.10 and Attempt-To-Finalize-Round(r) is described in Algorithm 5.11.

Algorithm 5.10

Best-Final-Candidate(r)

Algorithm 5.11

Attempt-To-Finalize-Round(r)

5.4. Block Finalization

DEFINITION 5.27. *A Polkadot relay chain node n should consider block B as **finalized** if any of the following criteria holds for $B' \geqslant B$:*

- $V^{r,\mathrm{pc}}_{\mathrm{obs}(n)}(B') > 2/3|\mathbb{V}_{B'}|$.
- *it receives a $M^{r,\mathrm{Fin}}_{v}(B')$ message in which $J^r(B)$ justifies the finalization (according to Definition 5.25).*
- *it receives a block data message for B' with $\mathrm{Just}(B')$ defined in Section 3.3.1.2 which justifies the finalization.*

for

- any round r if the node n is *not* a GRANDPA voter.
- only for rounds r for which the the node n has invoked Algorithm 5.9 if n is a GRANDPA voter.

Note that all Polkadot relay chain nodes are supposed to listen to GRANDPA finalizing messages regardless if

they are GRANDPA voters.

Appendix A

Cryptographic Algorithms

A.1. Hash Functions

A.2. BLAKE2

BLAKE2 is a collection of cryptographic hash functions known for their high speed. their design closely resembles BLAKE which has been a finalist in SHA-3 competition.

Polkadot is using Blake2b variant which is optimized for 64bit platforms. Unless otherwise specified, Blake2b hash function with 256bit output is used whenever Blake2b is invoked in this document. The detailed specification and sample implementations of all variants of Blake2 hash functions can be found in RFC 7693 [SA15].

A.3. Randomness

A.4. VRF

A.5. Cryptographic Keys

Various types of keys are used in Polkadot to prove the identity of the actors involved in Polkadot Protocols. To improve the security of the users, each key type has its own unique function and must be treated differently, as described by this section.

Definition A.1. ***Account key*** $(\mathbf{sk}^a, \mathbf{pk}^a)$ *is a key pair of type of either of schemes listed in Table A.1:*

Key scheme	*Description*
SR25519	*Schnorr signature on Ristretto compressed Ed25519 points as implemented in [Bur19]*
ED25519	*The standard ED25519 signature complying with [JL17]*
secp256k1	*Only for outgoing transfer transactions*

Table A.1. *List of public key scheme which can be used for an account key*

Account key can be used to sign transactions among other accounts and blance-related functions.

There are two prominent subcategories of account keys namely "stash keys" and "controller keys", each being used for a different function as described below.

Definition A.2. *The* ***Stash key*** *is a type of an account key that holds funds bonded for staking (described in Section A.5.1) to a particular controller key (defined in Definition A.3). As a result, one may actively participate with a stash key keeping the stash key offline in a secure location. It can also be used to designate a Proxy account to vote in governance proposals, as described in A.5.2. The Stash key holds the majority of the users' funds and should neither be shared with anyone, saved on an online device, nor used to submit extrinsics.*

DEFINITION A.3. *The **Controller key** is a type of account key that acts on behalf of the Stash account. It signs transactions that make decisions regarding the nomination and the validation of other keys. It is a key that will be in the direct control of a user and should mostly be kept offline, used to submit manual extrinsics. It sets preferences like payout account and commission, as described in A.5.4. If used for a validator, it certifies the session keys, as described in A.5.5. It only needs the required funds to pay transaction fees [key needing fund needs to be defined].*

Keys defined in Definitions A.1, A.2 and A.3 are created and managed by the user independent of the Polkadot implementation. The user notifies the network about the used keys by submitting a transaction, as defined in A.5.2 and A.5.5 respectively.

DEFINITION A.4. ***Session keys** are short-lived keys that are used to authenticate validator operations. Session keys are generated by Polkadot RE and should be changed regularly due to security reasons. Nonetheless, no validity period is enforced by Polkadot protocol on session keys. Various types of keys used by Polkadot RE are presented in Table A.2:*

Protocol	*Key scheme*
GRANDPA	*ED25519*
BABE	*SR25519*
I'm Online	*SR25519*
Parachain	*SR25519*

***Table A.2.** List of key schemes which are used for session keys depending on the protocol*

Session keys must be accessible by certain Runtime Environment APIs defined in Appendix E. Session keys are *not* meant to control the majority of the users' funds and should only be used for their intended purpose. [key managing fund need to be defined]

A.5.1. Holding and staking funds

To be specced

A.5.2. Creating a Controller key

To be specced

A.5.3. Designating a proxy for voting

To be specced

A.5.4. Controller settings

To be specced

A.5.5. Certifying keys

Session keys should be changed regularly. As such, new session keys need to be certified by a controller key before putting in use. The controller only needs to create a certificate by signing a session public key and broadcastg

this certificate via an extrinsic. [spec the detail of the data structure of the certificate etc.]

Appendix B

Auxiliary Encodings

B.1. SCALE Codec

Polkadot RE uses *Simple Concatenated Aggregate Little-Endian" (SCALE) codec* to encode byte arrays as well as other data structures. SCALE provides a canonical encoding to produce consistent hash values across their implementation, including the Merkle hash proof for the State Storage.

Definition B.1. *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:

$$\mathrm{Enc}_{\mathrm{SC}}(A) := \mathrm{Enc}^{\mathrm{Len}}_{\mathrm{SC}}(\|A\|)||A$$

where $\mathrm{Enc}^{\mathrm{Len}}_{\mathrm{SC}}$ is defined in Definition B.8.

Definition B.2. *The **SCALE codec** for **Tuple** T such that:*

$$T := (A_1, ..., A_n)$$

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

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

In case of a tuple (or struct), the knowledge of the shape of data is not encoded even though it is necessary for decoding. The decoder needs to derive that information from the context where the encoding/decoding is happenning.

Definition B.3. *We define a **varying data** type to be an ordered set of data types*

$$\mathcal{T} = \{T_1, ..., T_n\}$$

A value $\boldsymbol{A}$ of varying date type is a pair $(A_{\mathrm{Type}}, A_{\mathrm{Value}})$ where $A_{\mathrm{Type}} = T_i$ for some $T_i \in \mathcal{T}$ and A_{Value} is its value of type T_i. We define $\mathrm{idx}(T_i) = i - 1$.

*In particular, we define **optional type** to be $\mathcal{O} = \{\mathrm{None}, T_2\}$ for some data type T_2 where $\mathrm{idx}(\mathrm{None}) = 0$ (None, ϕ) is the only possible value, when the data is of type None and a codec value is one byte of 0 value.*

Definition B.4. *Scale coded for value $\boldsymbol{A = (A_{\mathrm{Type}}, A_{\mathrm{Value}})}$ of varying data type $\mathcal{T} = \{T_1, ..., T_n\}$*

$$\mathrm{Enc}_{\mathrm{SC}}(A) := \mathrm{Enc}_{\mathrm{SC}}(\mathrm{Idx}(A_{\mathrm{Type}}))||\mathrm{Enc}_{\mathrm{SC}}(A_{\mathrm{Value}})$$

Where Idx *is encoded in a fixed length integer determining the type of A.*

In particular, for the optional type defined in Definition B.3, we have:

$$\text{Enc}_{\text{SC}}((\text{None}, \phi)) := 0_{\mathbb{B}_1}$$

SCALE codec does not encode the correspondence between the value of Idx defined in Definition B.4 and the data type it represents; the decoder needs prior knowledge of such correspondence to decode the data.

DEFINITION B.5. *The **SCALE codec** for **sequence** S such that:*

$$S := A_1, ..., A_n$$

*where A_i's are values of **the same type** (and the decoder is unable to infer value of n from the context) is defined as:*

$$\text{Enc}_{\text{SC}}(S) := \text{Enc}^{\text{Len}}_{\text{SC}}(\|S\|)\text{Enc}_{\text{SC}}(A_1)|\text{Enc}_{\text{SC}}(A_2)|...|\,\text{Enc}_{\text{SC}}(A_n)$$

*where $\text{Enc}^{\text{Len}}_{\text{SC}}$ is defined in Definition B.8. SCALE codec for **dictionary** or **hashtable** D with key-value pairs (k_i, v_i)s such that:*

$$D := \{(k_1, v_1), ..., (k_1, v_n)\}$$

is defined the SCALE codec of D as a sequence of key value pairs (as tuples):

$$\text{Enc}_{\text{SC}}(D) := \text{Enc}^{\text{Len}}_{\text{SC}}(\|D\|)\text{Enc}_{\text{SC}}((k_1, v_1))|\text{Enc}_{\text{SC}}((k_2, v_2))|...|\,\text{Enc}_{\text{SC}}((k_n, v_n))$$

DEFINITION B.6. *The **SCALE codec** for **boolean value** b defined as a byte as follows:*

$$\begin{aligned} \text{Enc}_{\text{SC}}:\ & \{\text{False}, \text{True}\} \to \mathbb{B}_1 \\ & b \to \begin{cases} 0 & b = \text{False} \\ 1 & b = \text{True} \end{cases} \end{aligned}$$

DEFINITION B.7. *The **SCALE codec, Enc_{SC}** for other types such as fixed length integers not defined here otherwise, is equal to little endian encoding of those values defined in Definition 1.7.*

B.1.1. Length Encoding

SCALE Length encoding is used to encode integer numbers of variying sizes prominently in an encoding length of arrays:

DEFINITION B.8. ***SCALE Length Encoding, $\text{Enc}^{\text{Len}}_{\text{SC}}$** also known as compact encoding of a non-negative integer number n is defined as follows:*

$$\begin{aligned} \text{Enc}^{\text{Len}}_{\text{SC}}:\ & \mathbb{N} \to \mathbb{B} \\ & n \to b := \begin{cases} l_1 & 0 \leqslant n < 2^6 \\ i_1\, i_2 & 2^6 \leqslant n < 2^{14} \\ j_1\, j_2\, j_3 & 2^{14} \leqslant n < 2^{30} \\ k_1\, k_2 \ldots k_m & 2^{30} \leqslant n \end{cases} \end{aligned}$$

in where the least significant bits of the first byte of byte array b are defined as follows:

$$\begin{aligned} l_1^1\, l_1^0 &= 00 \\ i_1^1\, i_1^0 &= 01 \\ j_1^1\, j_1^0 &= 10 \\ k_1^1\, k_1^0 &= 11 \end{aligned}$$

and the rest of the bits of b store the value of n in little-endian format in base-2 as follows:

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

such that:

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

B.2. Frequently SCALEd Object

In this section, we will specify the objects which are frequently used in transmitting data between PDRE, Runtime and other clients and their SCALE encodings.

B.2.1. Result

[Spec Result Object]

B.2.2. Error

[Spec Error Object]

B.3. 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 B.9. *Suppose that* $\mathrm{PK} = (k_1, \ldots, k_n)$ *is a sequence of nibbles, then*
$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) :=$

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

Appendix C

Genesis State Specification

The genesis state represents the intial state of Polkadot state storage as a set of key-value pairs, which can be retrieved from [Fou20]. While each of those key/value pairs offer important identifyable information which can be used by the Runtime, from Polkadot RE points of view, it is a set of arbitrary key-value pair data as it is chain and network dependent. Except for the `:code` described in Section 3.1.1 which needs to be identified by the Polkadot RE to load its content as the Runtime. The other keys and values are unspecifed and its usage depends on the chain respectively its corresponding Runtime. The data should be inserted into the state storage with the `set_storage` RE API, as defined in Section E.1.1.

As such, Polkadot does not defined a formal genesis block. Nonetheless for the complatibilty reasons in several algorithms, Polkadot RE defines the *genesis header* according to Definition C.1. By the abuse of terminalogy, "*genesis block*" refers to the hypothetical parent of block number 1 which holds genisis header as its header.

Definition C.1. *The Polkadot genesis header is a data structure conforming to block header format described in section 3.6. It contains the values depicted in Table C.1:*

Block header field	*Genesis Header Value*
parent_hash	*0*
number	*0*
`state_root`	*Merkle hash of the state storage trie as defined in Definition 2.12 after inserting the genesis state in it.*
extrinsics_root	*0*
digest	*0*

Table C.1. *Genesis header values*

Appendix D
Network Messages

In this section, we will specify various types of messages which Polkadot RE receives from the network. Furthermore, we also explain the appropriate responses to those messages.

Definition D.1. *A **network message** is a byte array, **M** of length $\|M\|$ such that:*

$$\begin{array}{cc} M_1 & \text{Message Type Indicator} \\ M_2 ... M_{\|M\|} & \text{Enc}_{\text{SC}}(\text{MessageBody}) \end{array}$$

The body of each message consists of different components based on its type. The different possible message types are listed below in Table D.1. We describe the sub-components of each message type individually in Section D.1.

M_1	Message Type	Description
0	Status	D.1.1
1	Block Request	D.1.2
2	Block Response	D.1.3
3	Block Announce	D.1.4
4	Transactions	D.1.5
5	Consensus	D.1.6
6	Remote Call Request	
7	Remote Call Response	
8	Remote Read Request	
9	Remote Read Response	
10	Remote Header Request	
11	Remote Header Response	
12	Remote Changes Request	
13	Remote Changes Response	
14	FinalityProofRequest	
15	FinalityProofResponse	
255	Chain Specific	

Table D.1. List of possible network message types.

D.1. Detailed Message Structure

This section disucsses the detailed structure of each network message.

D.1.1. Status Message

A *Status* Message represented by M_S is sent after a connection with a neighbouring node is established and has the following structure:

$$M_S := \text{Enc}_{\text{SC}}(v, r, N_B, \text{Hash}_B, \text{Hash}_G, C_S)$$

Where:

v:	Protocol version	32 bit integer
v_{min}:	Minimum supported version	32 bit integer
r:	Roles	1 byte
N_B:	Best Block Number	64 bit integer
Hash_B	Best block Hash	$\mathbb{B}_{32}$
Hash_G	Genesis Hash	$\mathbb{B}_{32}$
C_S	Chain Status	Byte array

In which, Role is a bitmap value whose bits represent different roles for the sender node as specified in Table D.2:

Value	Binary representation	Role
0	00000000	No network
1	00000001	Full node, does not participate in consensus
2	00000010	Light client node
4	00000100	Act as an authority

Table D.2. Node role representation in the status message.

D.1.2. Block Request Message

A Block request message, represented by M_{BR}, is sent to request block data for a range of blocks from a peer and has the following structure:

$$M_{\text{BR}} := \text{Enc}_{\text{SC}}(\text{id}, A_B, S_B, \text{Hash}_E, d, \text{Max})$$

where:

id:	Unique request id	32 bit integer
A_B:	Requested data	1 byte
S_B:	Starting Block	Varying $\{\mathbb{B}_{32}, 64\text{bit integer}\}$
Hash_E	End block Hash	$\mathbb{B}_{32}$ optional type
d	Block sequence direction	1 byte
Max	Maximum number of blocks to return	32 bit integer optional type

in which

– A_B, the requested data, is a bitmap value, whose bits represent the part of the block data requested, as explained in Table D.3:

Value	Binary representation	Requested Attribute
1	00000001	Block header
2	00000010	Block Body
4	00000100	Receipt
8	00001000	Message queue
16	00010000	Justification

Table D.3. Bit values for block attribute A_B, to indicate the requested parts of the data.

– S_B is SCALE encoded varying data type (see Definition B.4) of either $\mathbb{B}_{32}$ representing the block hash, H_B, or 64bit integer representing the block number of the starting block of the requested range of blocks.

– Hash_E is optionally the block hash of the last block in the range.

– d is a flag; it defines the direction on the block chain where the block range should be considered (starting with the starting block), as follows

$$d = \begin{cases} 0 & \text{child to parent direction} \\ 1 & \text{parent to child direction} \end{cases}$$

Optional data type is defined in Definition B.3.

D.1.3. Block Response Message

A *block response message* represented by M_{BS} is sent in a response to a requested block message (see Section D.1.2). It has the following structure:

$$M_{\text{BS}} := \text{Enc}_{\text{SC}}(\text{id}, D)$$

where:

id:	Unique id of the requested response was made for	32 bit integer
D:	Block data for the requested sequence of Block	Array of block data

In which block data is defined in Definition D.2.

Definition D.2. ***Block Data*** *is defined as the followig tuple:[Block Data definition should go to block format section]*

$$(H_B, \text{Header}_B, \text{Body}, \text{Receipt}, \text{MessageQueue}, \text{Justification})$$

Whose elements, with the exception of H_B, are all of the following *optional type* (see Definition B.3) and are defined as follows:

H_B:	Block header hash	$\mathbb{B}_{32}$
Header_B:	Block header	5-tuple (Definition 3.6)
Body	Array of extrinsics	Array of Byte arrays (Section 3.2)
Receipt	Block Receipt	Byte array
Message Queue	Block message queue	Byte array
Justification	Block Justification	Byte array

D.1.4. Block Announce Message

A *block announce message* represented by M_{BA} is sent when a node becomes aware of a new complete block on the network and has the following structure:

$$M_{\text{BA}} := \text{Enc}_{\text{SC}}(\text{Header}_B)$$

Where:

Header_B:	Header of new block B	5-tuple header (Definition 3.6)

D.1.5. Transactions

The transactions Message is represented by M_T and is defined as follows:

$$M_T := \text{Enc}_{\text{SC}}(C_1, ..., C_n)$$

in which:

$$C_i := \text{Enc}_{\text{SC}}(E_i)$$

Where each E_i is a byte array and represents a sepearate extrinsic. Polkadot RE is indifferent about the content of an extrinsic and treats it as a blob of data.

D.1.6. Consensus Message

A *consensus message* represented by M_C is sent to communicate messages related to consensus process:

$$M_C := \text{Enc}_{\text{SC}}(E_{\text{id}}, D)$$

Where:

E_{id}:	The consensus engine unique identifier	$\mathbb{B}_4$
D	Consensus message payload	$\mathbb{B}$

in which

$$E_{\text{id}} := \begin{cases} \text{"BABE"} & \text{For messages related to BABE protocol refered to as } E_{\text{id}}(\text{BABE}) \\ \text{"FRNK"} & \text{For messages related to GRANDPA protocol referred to as } E_{\text{id}}(\text{FRNK}) \end{cases}$$

The network agent should hand over D to approperiate consensus engine which identified by E_{id}.

Appendix E

Runtime Environment API

The Runtime Environment API is a set of functions that Polkadot RE exposes to Runtime to access external functions needed for various reasons, such as the Storage of the content, access and manipulation, memory allocation, and also efficiency. We introduce Notation E.1 to emphasize that the result of some of the API functions depends on the content of state storage.

Notation E.1. *By $\mathcal{RE}_B$ we refer to the API exposed by Polkadot RE which interact, manipulate and response based on the state storage whose state is set at the end of the execution of block B.*

The functions are specified in each subsequent subsection for each category of those functions.

E.1. Storage

E.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.

E.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.

E.1.3. `ext_blake2_256_enumerated_trie_root`

Given an array of byte arrays, it arranges them in a Merkle trie, defined in Section 2.1.4, where the key under which the values are stored is the 0-based index of that value in the array. It computes and returns the root hash of the constructed trie.

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`.

E.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.

E.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.

E.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.

E.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.

E.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 the specified chunk starting at the offset into 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 of 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.

E.1.9. `ext_set_child_storage`

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

Prototype:

```
(func $ext_set_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32) (param $key_data i32)
  (param $key_len i32) (param $value_data i32) (param $value_len i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key. This key **must** be prefixed with `:child_storage:default:`
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `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.

E.1.10. `ext_clear_child_storage`

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

Prototype:

```
(func $ext_clear_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32)
  (param $key_data i32) (param $key_len i32))
```

Arguments:

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

E.1.11. `ext_exists_child_storage`

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

Prototype:

```
(func $ext_exists_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32)
  (param $key_data i32) (param $key_len i32) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the key 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 child storage or 0 if the child storage does not contain an entry with the given key.

E.1.12. `ext_get_allocated_child_storage`

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

Prototype:

```
(func $ext_get_allocated_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32) (param $key_data i32)
  (param $key_len i32) (param $written_out) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.

- `key_data`: a memory address pointing at the buffer of the byte array containing the key value.
- `key_len`: the length of the key 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 stores $2^{32} - 1$.
- `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.

E.1.13. `ext_get_child_storage_into`

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

Prototype:

```
(func $ext_get_child_storage_into
  (param $storage_key_data i32) (param $storage_key_len i32)
  (param $key_data i32) (param $key_len i32) (param $value_data i32)
  (param $value_len i32) (param $value_offset i32) (result i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.
- `key_data`: a memory address pointing at the buffer of 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.

E.1.14. `ext_kill_child_storage`

Given a byte array, this function removes all entries of the child storage whose child storage key is specified in the array.

Prototype:

```
(func $ext_kill_child_storage
  (param $storage_key_data i32) (param $storage_key_len i32))
```

Arguments:

- `storage_key_data`: a memory address pointing at the buffer of the byte array containing the child storage key.
- `storage_key_len`: the length of the child storage key byte array in number of bytes.

E.1.15. Memory

E.1.15.1. `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.

E.1.15.2. `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.

E.1.15.3. Input/Output

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

E.1.16. Cryptograhpic Auxiliary Functions

E.1.16.1. `ext_blake2_256`

Computes the Blake2b 256bit 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 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 Blake2b hash of the data.

E.1.16.2. `ext_keccak_256`

Computes the Keccak-256 hash of a given byte array.

Prototype:

```
(func $ext_keccak_256
      (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer 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 Keccak-256 hash of the data.

E.1.16.3. `ext_twox_128`

Computes the *xxHash64* algorithm (see [Col19]) twice initiated with seeds 0 and 1 and applied on a given byte array and outputs the concatenated result.

Prototype:

```
(func $ext_twox_128
       (param $data i32) (param $len i32) (param $out i32))
```

Arguments:

- `data`: a memory address pointing at the buffer 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 16-byte byte array containing $xxhash64_0(\texttt{data})||xxhash64_1(\texttt{data})$ where $xxhash64_i$ is the xxhash64 function initiated with seed i as a 64bit unsigned integer.

E.1.16.4. `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.

E.1.16.5. `ext_sr25519_verify`

Given a message signed by the SR25519 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_sr25519_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 SR25519 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.

E.1.16.6. To be Specced

- `ext_twox_256`

E.1.17. Offchain Worker

The Offchain Workers allow the execution of long-running and possibly non-deterministic tasks (e.g. web requests, encryption/decryption and signing of data, random number generation, CPU-intensive computations, enumeration/aggregation of on-chain data, etc.) which could otherwise require longer than the block execution time. Offchain Workers have their own execution environment. This separation of concerns is to make sure that the block production is not impacted by the long-running tasks.

All data and results generated by Offchain workers are unique per node and nondeterministic. Information can be propagated to other nodes by submitting a transaction that should be included in the next block. As Offchain workers runs on their own execution environment they have access to their own separate storage. There are two different types of storage available which are defined in Definitions E.2 and E.3.

DEFINITION E.2. ***Persistent storage*** *is non-revertible and not fork-aware. It means that any value set by the offchain worker is persisted even if that block (at which the worker is called) is reverted as non-canonical (meaning that the block was surpassed by a longer chain). The value is available for the worker that is re-run at the new (different block with the same block number) and future blocks. This storage can be used by offchain workers to handle forks and coordinate offchain workers running on different forks.*

DEFINITION E.3. ***Local storage*** *is revertible and fork-aware. It means that any value set by the offchain worker triggered at a certain block is reverted if that block is reverted as non-canonical. The value is NOT available for the worker that is re-run at the next or any future blocks.*

DEFINITION E.4. ***HTTP status codes*** *that can get returned by certain Offchain HTTP functions.*

- ***0****: the specified request identifier is invalid.*
- ***10****: the deadline for the started request was reached.*
- ***20****: an error has occurred during the request, e.g. a timeout or the remote server has closed the connection. On returning this error code, the request is considered destroyed and must be reconstructed again.*
- ***100..999****: the request has finished with the given HTTP status code.*

E.1.17.1. `ext_is_validator`

Returns if the local node is a potential validator. Even if this function returns 1, it does not mean that any keys are configured and that the validator is registered in the chain.

Prototype:

```
(func $ext_is_validator
      (result i32))
```

Arguments:

- `result`: an i32 integer which is equal to 1 if the local node is a potential validator or a equal to 0 if it is not.

E.1.17.2. `ext_submit_transaction`

Given an extrinsic as a SCALE encoded byte array, the system decodes the byte array and submits the extrinsic in the inherent pool as an extrinsic to be included in the next produced block.

Prototype:

```
(func $ext_submit_transaction
      (param $data i32) (param $len i32) (result i32))
```

Arguments:

- `data`: a pointer to the buffer containing the byte array storing the encoded extrinsic.
- `len`: an `i32` integer indicating the size of the encoded extrinsic.
- `result`: an integer value equal to 0 indicates that the extrinsic is successfully added to the pool or a nonzero value otherwise.

E.1.17.3. `ext_network_state`

Returns the SCALE encoded, opaque information about the local node's network state. This information is fetched by calling into `libp2p`, which *might* include the `PeerId` and possible `Multiaddress(-es)` by which the node is publicly known by. Those values are unique and have to be known by the node individually. Due to its opaque nature, it's unknown whether that information is available prior to execution.

Prototype:

```
(func $ext_network_state
      (param $written_out i32)(result i32))
```

Arguments:

- `written_out`: a pointer to the 4-byte buffer where the size of the opaque network state gets written to.
- `result`: a pointer to the buffer containing the SCALE encoded network state. This includes none or one `PeerId` followed by none, one or more IPv4 or IPv6 `Multiaddress(-es)` by which the node is publicly known by.

E.1.17.4. `ext_timestamp`

Returns current timestamp.

Prototype:

```
(func $ext_timestamp
      (result i64))
```

Arguments:

- `result`: an i64 integer indicating the current UNIX timestamp as defined in Definition 1.10.

E.1.17.5. `ext_sleep_until`

Pause the execution until 'deadline' is reached.

Prototype:

```
(func $ext_sleep_until
      (param $deadline i64))
```

Arguments:

- `deadline`: an i64 integer specifying the UNIX timestamp as defined in Definition 1.10.

E.1.17.6. ext_random_seed

Generates a random seed. This is a truly random non deterministic seed generated by the host environment.

Prototype:

```
(func $ext_random_seed
      (param $seed_data i32))
```

Arguments:

- `seed_data`: a memory address pointing at the beginning of a 32-byte byte array containing the generated seed.

E.1.17.7. ext_local_storage_set

Sets a value in the local storage. This storage is not part of the consensus, it's only accessible by the offchain worker tasks running on the same machine and is persisted between runs.

Prototype:

```
(func $ext_local_storage_set
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $value i32) (param $value_len i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition E.2 and a value equal to 2 for local storage as defined in Definition E.3.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.
- `value`: a pointer to the buffer containg the value.
- `value_len`: an i32 integer indicating the size of the value.

E.1.17.8. ext_local_storage_compare_and_set

Sets a new value in the local storage if the condition matches the current value.

Prototype:

```
(func $ext_local_storage_compare_and_set
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $old_value i32) (param $old_value_len) (param $new_value i32)
      (param $new_value_len) (result i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition E.2 and a value equal to 2 for local storage as defined in Definition E.3.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.

- `old_value`: a pointer to the buffer containing the current value.
- `old_value_len`: an i32 integer indicating the size of the current value.
- `new_value`: a pointer to the buffer containing the new value.
- `new_value_len`: an i32 integer indicating the size of the new value.
- `result`: an i32 integer equal to 0 if the new value has been set or a value equal to 1 if otherwise.

E.1.17.9. `ext_local_storage_get`

Gets a value from the local storage.

Prototype:

```
(func $ext_local_storage_set
      (param $kind i32) (param $key i32) (param $key_len i32)
      (param $value_len i32) (result i32))
```

Arguments:

- `kind`: an i32 integer indicating the storage kind. A value equal to 1 is used for a persistent storage as defined in Definition E.2 and a value equal to 2 for local storage as defined in Definition E.3.
- `key`: a pointer to the buffer containing the key.
- `key_len`: an i32 integer indicating the size of the key.
- `value_len`: an i32 integer indicating the size of the value.
- `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.

E.1.17.10. `ext_http_request_start`

Initiates a http request given by the HTTP method and the URL. Returns the id of a newly started request.

Prototype:

```
(func $ext_http_request_start
      (param $method i32) (param $method_len i32) (param $url i32)
      (param $url_len i32) (param $meta i32) (param $meta_len i32) (result i32))
```

Arguments:

- `method`: a pointer to the buffer containing the key.
- `method_len`: an i32 integer indicating the size of the method.
- `url`: a pointer to the buffer containing the url.
- `url_len`: an i32 integer indicating the size of the url.
- `meta`: a future-reserved field containing additional, SCALE encoded parameters.
- `meta_len`: an i32 integer indicating the size of the parameters.
- `result`: an i32 integer indicating the ID of the newly started request.

E.1.17.11. `ext_http_request_add_header`

Append header to the request. Returns an error if the request identifier is invalid, `http_response_wait` has already been called on the specified request identifier, the deadline is reached or an I/O error has happened (e.g. the remote has closed the connection).

Prototype:

```
(func $ext_http_request_add_header
      (param $request_id i32) (param $name i32) (param $name_len i32)
      (param $value i32) (param $value_len i32) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `name`: a pointer to the buffer containing the header name.
- `name_len`: an i32 integer indicating the size of the header name.
- `value`: a pointer to the buffer containing the header value.
- `value_len`: an i32 integer indicating the size of the header value.
- `result`: an i32 integer where the value equal to 0 indicates if the header has been set or a value equal to 1 if otherwise.

E.1.17.12. `ext_http_request_write_body`

Writes a chunk of the request body. Writing an empty chunk finalises the request. Returns a non-zero value in case the deadline is reached or the chunk could not be written.

Prototype:

```
(func $ext_http_request_write_body
      (param $request_id i32) (param $chunk i32) (param $chunk_len i32)
      (param $deadline i64) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `chunk`: a pointer to the buffer containing the chunk.
- `chunk_len`: an i32 integer indicating the size of the chunk.
- `deadline`: an i64 integer specifying the UNIX timestamp as defined in Definition 1.10. Passing '0' will block indefinitely.
- `result`: an i32 integer where the value equal to 0 indicates if the header has been set or a non-zero value if otherwise.

E.1.17.13. `ext_http_response_wait`

Blocks and waits for the responses for given requests. Returns an array of request statuses (the size is the same as number of IDs).

Prototype:

```
(func $ext_http_response_wait
      (param $ids i32) (param $ids_len i32) (param $statuses i32)
      (param $deadline i64))
```

Arguments:

- `ids`: a pointer to the buffer containing the started IDs.
- `ids_len`: an i32 integer indicating the size of IDs.
- `statuses`: a pointer to the buffer where the request statuses get written to as defined in Definition E.4. The lenght is the same as the length of `ids`.
- `deadline`: an i64 integer indicating the UNIX timestamp as defined in Definition 1.10. Passing '0' as deadline will block indefinitely.

E.1.17.14. `ext_http_response_headers`

Read all response headers. Returns a vector of key/value pairs. Response headers must be read before the response body.

Prototype:

```
(func $ext_http_response_headers
      (param $request_id i32) (param $written_out i32) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `written_out`: a pointer to the buffer where the size of the response headers gets written to.
- `result`: a pointer to the buffer containing the response headers.

E.1.17.15. `ext_http_response_read_body`

Reads a chunk of body response to the given buffer. Returns the number of bytes written or an error in case a deadline is reached or the server closed the connection. If '0' is returned it means that the response has been fully consumed and the `request_id` is now invalid. This implies that response headers must be read before draining the body.

Prototype:

```
(func $ext_http_response_read_body
      (param $request_id i32) (param $buffer i32) (param $buffer_len)
      (param $deadline i64) (result i32))
```

Arguments:

- `request_id`: an i32 integer indicating the ID of the started request.
- `buffer`: a pointer to the buffer where the body gets written to.
- `buffer_len`: an i32 integer indicating the size of the buffer.

- `deadline`: an i64 integer indicating the UNIX timestamp as defined in Definition 1.10. Passing '0' will block indefinitely.
- `result`: an i32 integer where the value equal to 0 indicateds a fully consumed response or a non-zero value if otherwise.

E.1.18. Sandboxing

E.1.18.1. 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`

E.1.19. Auxillary Debugging API

E.1.19.1. `ext_print_hex`

Prints out the content of the given buffer on the host's debugging console. Each byte is represented as a two-digit hexadecimal number.

Prototype:

```
(func $ext_print_hex
  (param $data i32) (parm $len i32))
```

Arguments:

- `data`: a pointer to the buffer containing the data that needs to be printed.
- `len`: an `i32` integer indicating the size of the buffer containing the data in bytes.

E.1.19.2. `ext_print_utf8`

Prints out the content of the given buffer on the host's debugging console. The buffer content is interpreted as a UTF-8 string if it represents a valid UTF-8 string, otherwise does nothing and returns.

Prototype:o

```
(func $ext_print_utf8
  (param $utf8_data i32) (param $utf8_len i32))
```

Arguments:

- `utf8_data`: a pointer to the buffer containing the utf8-encoded string to be printed.
- `utf8_len`: an `i32` integer indicating the size of the buffer containing the UTF-8 string in bytes.

E.1.20. Misc

E.1.20.1. To be Specced

- `ext_chain_id`

E.1.21. Block Production

E.2. Validation

Appendix F

Runtime Entries

F.1. List of Runtime Entries

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

```
(export "Core_version" (func $Core_version))
(export "Core_execute_block" (func $Core_execute_block))
(export "Core_initialize_block" (func $Core_initialize_block))
(export "Metadata_metadata" (func $Metadata_metadata))
(export "BlockBuilder_apply_extrinsic" (func $BlockBuilder_apply_extrinsic))
(export "BlockBuilder_finalize_block" (func $BlockBuilder_finalize_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_validators" (func $ParachainHost_validators))
(export "ParachainHost_duty_roster" (func $ParachainHost_duty_roster))
(export "ParachainHost_active_parachains"
        (func $ParachainHost_active_parachains))
(export "ParachainHost_parachain_status" (func $ParachainHost_parachain_status))
(export "ParachainHost_parachain_code" (func $ParachainHost_parachain_code))
(export "ParachainHost_ingress" (func $ParachainHost_ingress))
(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 "BabeApi_startup_data" (func $BabeApi_startup_data))
(export "BabeApi_epoch" (func $BabeApi_epoch))
(export "SessionKeys_generate_session_keys"
        (func $SessionKeys_generate_session_keys))
```

Snippet F.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.

F.2. Argument Specification

As a wasm functions, all runtime entries have the following prototype signature:

```
(func $generic_runtime_entry
  (param $data i32) (parm $len i32) (reslut i64))
```

where `data` points to the SCALE encoded paramaters sent to the function and `len` is the length of the data. `result` can similarly either point to the SCALE encoded data the function returns (See Sections 3.1.2.2 and 3.1.2.3).

In this section, we describe the function of each of the entries alongside with the details of the arguments and the return values for each one of these enteries.

F.2.1. `Core_version`

This entry receives no argument; it returns the version data encoded in ABI format described in Section 3.1.2.3 containing the following information:

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

F.2.2. `Core_execute_block`

Executes a full block by executing all exctrinsics included in it and update the state accordingly. Additionally, some integrity checks are executed such as validating if the parent hash is correct and that the transaction root represents the transactions. Internally, this function performs an operation similar to the process described in Algorithm 5.7, by calling `Core_initialize_block`, `BlockBuilder_apply_extrinsics` and `BlockBuilder_finalize_block`.

This function should be called when a fully complete block is available that is not actively being built on, such as blocks received from other peers.

Arguments:

- The entry accepts the *block data* defined in Definition D.2 as the only argument.

Return:

- A boolean value indicates if the execution was successful.

F.2.3. `Core_initialize_block`

Sets up the environment required for building a new block as described in Algorithm 5.7.

Arguments:

- The block header of the new block as defined in 3.6. The values $\boldsymbol{H_r}$, $\boldsymbol{H_e}$ and $\boldsymbol{H_d}$ are left empty.

Return:

- None.

F.2.4. `hash_and_length`

An auxilarry function which returns hash and encoding length of an extrinsics.

Arguments:

- A blob of an extrinsic.

Return:

- Pair of Blake2Hash of the blob as element of $\mathbb{B}_{32}$ and its length as 64 bit integer.

F.2.5. `BabeApi_epoch`

This entry is called to obtain the current configuration of BABE consensus protocol.

Arguments:

- $H_n(B)$: the block number at whose final state the epoch configuration should be obtained.

Return:
A tuple

$$(\mathcal{E}_n, s_0^n, \mathrm{sc}_n, A, \rho, \mathrm{Sec})$$

where:

$\mathcal{E}_n$:	epoch index (see Definition 5.5)	64-bit integer
s_0^n:	The index of the starting slot of $\mathcal{E}_n$	64-bit integer
sc_n:	Slot count of $\mathcal{E}_n$ (see Definition 5.5)	1 byte
A:	The list of authorities and their weights	Array of (P_A, W_A)
ρ	Randomness used in $\mathcal{E}_n$ (see Section 5.2.5)	$\mathbb{B}_{32}$
Sec	To be specced	Boolean

in which:

P_A:	The public key of authority A	$\mathbb{B}_{32}$
W_A:	The weight of the authority A	64 bit integer

F.2.6. `GrandpaApi_grandpa_authorities`

This entry fetches the list of GRANDPA authorities according to the genesis block and is used to initialize authority list defined in Definition 5.1, at genisis. Any future authority changes get tracked via Runtiem-to-consensus engine messages as described in Section 5.1.2.

F.2.7. `TaggedTransactionQueue_validate_transaction`

This entry is invoked against extrinsics submitted through the Transaction network message D.1.5 and indicates if the submitted blob represents a valid extrinsics applied to the specified block.

Arguments:

- $H_n(B)$: the block number whose final state is where the transaction should apply the system state.
- UTX: A byte array that contains the transaction.

Return:

A varying type Result object which has type of *TransactionValidity* in case no error occurs in course of its execution. TransactionValidity is of varying type described in the Table F.2:

Type Index	Data type	Description
0	Byte	Indicating invalid extrinsic and bearing the error code concerning the cause of invalidity of the transaction.
1	A Quin-tuple	Indicating whether the extrinsic is valid and providing guidance for Polkadot RE on how to proceed with the extrinsic (see below)
2	Byte	The Validity of the extrinsic cannot be determined

Table F.2. Type variation for the return value of `TaggedTransactionQueue_transaction_validity`.

In which the quintuple of type for valid extrinsics consists of the following parts:

$$(\text{priority}, \text{requires}, \text{provides}, \text{longevity}, \text{propagate})$$

Name	Description	Type
Priority	Determines the ordering of two transactions that have all their dependencies (required tags) satisfied.	64bit integer
Requires	List of tags specifying extrinsics which should be applied before the current exrinsics can be applied.	Array of Transaction Tags
Provides	Informs Runtime of the extrinsics depending on the tags in the list that can be applied after current extrinsics are being applied. Describes the minimum number of blocks for the validity to be correct	Array of Transaction Tags
Longevity	After this period, the transaction should be removed from the pool or revalidated.	64 bit integer
Propagate	A flag indicating if the transaction should be propagated to other peers.	Boolean

Table F.3. The quintuple provided by `TaggedTransactionQueue_transaction_validity` in the case the transaction is judged to be valid.

Note that if *Propagate* is set to `false` the transaction will still be considered for including in blocks that are authored on the current node, but will never be sent to other peers.

F.2.8. `BlockBuilder_apply_extrinsic`

Apply the extrinsic outside of the block execution function. This doesn't attempt to validate anything regarding the block, but it builds a list of transaction hashes.

Arguments:

- An extrinisic.

Return:

- The result from the attempt to apply extrinsic. On success, it returns an array of zero length (one byte zero value). On failure, it either returns a Dispatch error or an Apply error. An Apply error uses identifiers to indicate the specific error type.

  Dispatch error (`0x0001` prefix) byte array, contains the following information:

Name	Type	Description
module	unsigned 8 bit integer	Module index, matching the metadata module index
error	unsigend 8 bit integer	Module specific error value

Table F.4. Data format of the Dispatch error type

  Apply error (`0x01` prefix). A Validity error type contains additional data for specific error types.

Identifier	Type	Description
`0x00`	NoPermission	General error to do wth the permissions of the sender
`0x01`	BadState	General error to do with the state of the system in general
`0x02`	Validity	Any error to do with the transaction validity
`0x020000`	Call	The call of the transaction is not expected
`0x020001`	Payment	Inability to pay fees (e.g. account balance too low)
`0x020002`	Future	Transaction not yet being valid (e.g. nonce too high)
`0x020003`	Stale	Transaction being outdated (e.g. nonce too low)
`0x020004`	BadProof	Invalid transaction proofs (e.g. bad signature)
`0x020005`	AncientBirthBlock	The transaction birth block is ancient
`0x020006`	ExhaustsResources	Would exhaust the resources of current block the transaction might be valid
`0x020007`	Custom	Any other custom invalidity of unknown size
`0x020100`	CannotLookup	Could not lookup some information that is required to validate the transaction
`0x020101`	NoUnsignedValidator	No validator found for the given unsigned transaction
`0x020102`	Custom	Any other custom invalidity of unknown size

Table F.5. Identifiers of the Apply error type

F.2.9. `BlockBuilder_inherent_extrinsics`

Generate inherent extrinsics. The inherent data will vary from chain to chain.

Arguments:

- A INHERENTS-DATA structure as defined in 3.5.

Return:

- An array of extrinisic where each extrinsic is a variable byte array.

F.2.10. `BlockBuilder_finalize_block`

Finalize the block - it is up to the caller to ensure that all header fields are valid except for the state root.

Glossary

$\boldsymbol{P_n}$	a path graph or a path of n nodes, can be represented by sequences of $(v_1, ..., v_n)$ 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} . . . 10
$\mathbb{B}_n$	a set of all byte arrays of length n . . . 10
I	the little-endian representation of a non-negative interger, represented as $I = (B_n...B_0)_{256}$ . . . 10
B	a byte array $B = (b_0, b_1, ..., b_n)$ such that $b_i := B_i$ . . . 10
Enc_{LE}	$\begin{array}{ccc} \mathbb{Z}^+ & \to & \mathbb{B} \\ (B_n...B_0)_{256} & \mapsto & (B_0, B_1, ..., B_n) \end{array}$ . . . 10
C, blockchain	a blockchain C is a directed path graph. . . . 11
Block	a node of the graph in blockchain C and indicated by B . . . 11
Genesis Block	the unique sink of blockchain C . . . 11
Head	the source of blockchain C . . . 11
P	for any vertex (B_1, B_2) where $B_1 \to B_2$ we say B_2 is the parent of B_1 and we indicate it by $B_2 := P(B_1)$. 11
BT, block tree	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 . . . 11
PBT, Pruned BT	Pruned Block Tree refers to a subtree of the block tree obtained by eliminating all branches which do not contain the most recent finalized blocks, as defined in Definition 5.27. . . . 11
pruning	. . . 11
G	is the root of the block tree and B is one of its nodes. . . . 11
CHAIN(B)	refers to the path graph from G to B in (P)BT. . . . 11
head of C	defines the head of C to be B, formally noted as $B := \textsc{Head}(C)$. . . . 11
$\lvert C \rvert$	defines he length of C as a path graph . . . 11
SubChain(B', B)	If B' is another node on $\textsc{Chain}(B)$, then by $\textsc{SubChain}(B', B)$ we refer to the subgraph of $\textsc{Chain}(B)$ path graph which contains both B and B'. . . . 11
$\mathbb{C}_{B'}((P)\text{BT})$	is the set of all subchains of (P)BT rooted at B'. . . . 11
$\mathbb{C}$	the set of all chains of (P)BT, $\mathbb{C}_G((P)\text{BT})$ is denoted by $\mathbb{C}$((P)BT) or simply $\mathbb{C}$ . . . 11
LONGEST-CHAIN(BT)	the maximum chain given by the complete order over $\mathbb{C}$ . . . 11
LONGEST-PATH(BT)	the path graph of (P)BT which is the longest among all paths in (P)BT and has the earliest block arrival time as defined in Definition 5.10. . . . 11
DEEPEST-LEAF(BT)	the head of LONGEST-PATH(BT) . . . 11
StoredValue	the function retrieves the value stored under a specific key in the state storage and is formally defined as $\mathcal{K} \to \mathcal{V}$ $k \mapsto \begin{cases} v & \text{if (k,v) exists in state storage} \\ \phi & \text{otherwise} \end{cases}$. Here $\mathcal{K} \subset \mathbb{B}$ and $\mathcal{V} \subset \mathbb{B}$ are respectively the set of all keys and values stored in the state storage. . . . 13

Bibliography

[Bur19] Jeff Burdges. Schnorr VRFs and signatures on the Ristretto group. Technical Report, 2019.

[Col19] Yann Collet. Extremely fast non-cryptographic hash algorithm. Technical Report, -, http://cyan4973.github.io/xxHash/, 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.

[Fou20] Web3.0 Technologies Foundation. Polkadot Genisis State. Technical Report, https://github.com/w3f/polkadot-spec/blob/master/genesis-state/, 2020.

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

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). In *Internet Research Task Force, Crypto Forum Research Group, RFC*, volume 8032. 2017.

[lab19] Protocol labs. Libp2p Specification. Technical Report, Protocol labs, https://github.com/libp2p/specs, 2019.

[LJ17] Ilari Liusvaara and Simon Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). 2017.

[Per18] Trevor Perrin. The Noise Protocol Framework. Technical Report, https://noiseprotocol.org/noise.html, 2018.

[SA15] Markku Juhani Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). ⟨keepcase|RFC⟩ 7693, -, https://tools.ietf.org/html/rfc7693, 2015.

[Ste19] Alistair Stewart. GRANDPA: A Byzantine Finality Gadget. 2019.

[Tec19] Parity Technologies. Substrate Reference Documentation. Rust ⟨keepcase | Doc⟩, Parity Technologies, https://substrate.dev/rustdocs/, 2019.

Index