Polkadot Protocol Specification

Web 3.0 Technologies Foundation, Fabio Lama, Florian Franzen, Syed Hosseini

Version 0.2.0

Table of Contents

This specification is **Work-In-Progress** and any content, structure, design and/or hyper/anchor-link **is subject to change**.

Host Specification

Description of the Host-side of the Polkadot Protocol

Chapter 1. Preliminaries

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 *Host*. The Runtime comprises the state transition logic for the Polkadot protocol and is designed and be upgradable via the consensus engine without requiring hard forks of the blockchain. The Polkadot Host provides the necessary functionality for the Runtime to execute its state transition logic, such as an execution environment, I/O, consensus and network interoperability between parachains. The Polkadot Host is planned to be stable and mostly static for the lifetime duration of the Polkadot protocol, the goal being that most changes to the protocol are primarily conducted by applying Runtime updates and not having to coordinate with network participants on manual software updates.

With the current document, we aim to specify the Polkadot Host 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 the Polkadot Host interacts with the state transition function, i.e. Runtime. Following, we specify the input messages triggering the state transition and the system behavior. In Chapter 4, we specify the communication protocols and network messages required for the Polkadot Host to communicate with other nodes in the network, such as exchanging blocks and consensus messages. 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 Chapter 12. A Polkadot Host implementation which conforms with this part of the specification should successfully be able to sync its states with the Polkadot network.

1.1. Bootstrapping

This section provides an overview over the tasks a Polkadot Host needs to perform in order to join and participate in the Polkadot network. While this chapter does not go into any new specifications of the protocol, it has been included to provide implementors with a pointer to what these steps are and where they are defined. In short, the following steps should be taken by all bootstrapping nodes:

1. The node must populate the state storage with the official genesis state, clarifier further in Chapter 12.
2. The node should maintain a set of around 50 active peers at any time. New peers can be found using the discovery protocols (Section 4.4)
3. The node should open and maintain the various required streams (Section 4.7) with each of its active peers.

4. Furthermore, the node should send block requests (Section 4.8.2) to these peers to receive all blocks in the chain and execute each of them.
5. The node should exchange neighbor packets (Section 4.8.6.1).

Validator nodes should take the following, additional steps.

1. Verify that the Host's session key is included in the current Epoch's authority set (Section 5.1.1).
2. Run the BABE lottery (Section 5.2) and wait for the next assigned slot in order to produce a block.
3. Gossip any produced blocks to all connected peers (Section 4.8.1).
4. Run the catch-up protocol (Section 5.4.1) to make sure that the node is participating in the current round and not a past round.
5. Run the GRANDPA rounds protocol (Section 5.3).

1.2. Definitions

Definition 1. Discrete State Machine (DSM)

A **Discrete State Machine (DSM)** is a state transition system that admits a starting state and whose set of states and set of transitions are countable. 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 2. Path Graph

A **path graph** or a **path** of n nodes formally referred to as 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 $(v_1, \text{ellipsis}, v_n)$ where $e_i = (v_i, v_{i+1})$ for $1 \leq i \leq n-1$ is the edge which connect v_i and v_{i+1}.

Definition 3. *Blockchain*

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, which is the **child** of B_2, respectively. We indicate that by:

$$B_2 := P(B_1)$$

The parent refers to the child by its hash value (Definition 28), making the path graph tamper proof since any modifications to the child would result in its hash value being changed.

The term "blockchain" can also be used as a way to refer to the network or system that interacts or maintains the directed path graph.

Definition 4. *Unix Time*

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 86'400 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 5. *Block*

The **block tree** of a blockchain, denoted by BT is the union of all different versions of the blockchain observed by the Polkadot Host such that every 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 (Section 5.3).

Definition 6. *Pruned Block Tree*

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 (Definition 95). By **pruning**, we refer to the procedure of $BT \leftarrow \text{PBT}$. When there is no risk of ambiguity and is safe to prune BT, we use BT to refer to PBT.

Definition 7 gives the means to highlight various branches of the block tree.

Definition 7. *Subchain*

Let G be the root of the block tree and B be one of its nodes. By $\text{Chain}(B)$, we refer to the path graph from G to B in BT. Conversely, for a chain $C = \text{Chain}(B)$, we define **the head of** C to be B, formally noted as $B := \overline{C}$. We define $|C|$, the length of C as a path graph.

If B' is another node on $\text{Chain}(B)$, then by $\text{SubChain}(B', B)$ we refer to the subgraph of $\text{Chain}(B)$ path graph which contains B and ends at B' and by $|\text{SubChain}(B', B)|$ we refer to its length.

Accordingly, $\mathbb{C}_{B'}(BT)$ is the set of all subchains of BT rooted at B'. The set of all chains of BT, $\mathbb{C}_G(BT))$ is denoted by $\mathbb{C}(BT)$ or simply $\mathbb{C}$, for the sake of brevity.

Definition 8. *Longest Chain*

We define the following complete order over $\mathbb{C}$ as follows. For chains $C_1, C_2 \in \mathbb{C}$ we have that $C_1 > C_2$ if either $|C_1| > |C_2|$ or $|C_1| = |C_2|$.

If $|C_1| = |C_2|$ we say $C_1 > C_2$ if and only if the block arrival time (Definition 75) of $\overline{C}_1$ is less than the block arrival time of $\overline{C}_2$, from the *subjective perspective* of the Host. We define the $\text{Longest-Chain}(BT)$ to be the maximum chain given by this order.

Definition 9. *Longest Path*

$\text{Longest-Path}(BT)$ returns the path graph of BT which is the longest among all paths in BT and has the earliest block arrival time (Definition 75). $\text{Deepest-Leaf}(BT)$ returns the head of $\text{Longest-Path}(BT)$ 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 10. *Descendant and Ancestor*

We say B is **descendant** of B', formally noted as $B > B'$, if $(|B| > |B'|) \in C$. Respectively, we say that B' is an **ancestor** of B, formally noted as $B < B'$, if $(|B| < |B'|) \in C$.

Chapter 2. State Specification

2.1. State Storage and Storage Trie

For storing the state of the system, Polkadot Host 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 (Section 11.2.1).

2.1.1. Accessing System Storage

The Polkadot Host implements various functions to facilitate access to the system storage for the Runtime (Section 3.1). Here we formalize the access to the storage when it is being directly accessed by the Polkadot Host (in contrast to Polkadot runtime).

Definition 11. Stored Value

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

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

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. $\mathcal{V}$ can be an empty value.

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 ***State Trie*** or just ***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 *merkle root* (Section 2.1.4) of the state, H_r (Definition 28), whose purpose is to authenticate the validity of the state database. Thus, the Polkadot Host 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 Host implementations.

The trie is a *radix-16* tree (Definition 12). 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.

Definition 12. *Radix-r Tree*

A ***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.

When traversing the trie to a specific node, its key can be reconstructed by concatenating the subsequences of the keys 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 way consistent with the trie structure. Because each node in the trie has at most 16 children, we represent the key as a sequence of 4-bit nibbles:

Definition 13. *Key Encode*

For the purpose of labeling the branches of the trie, the key k is encoded to k_{enc} using $\mathsf{KeyEncode}$ functions:

$$k_{\text{enc}} := \left(k_{\text{enc}_1}, \text{ellipsis}, k_{\text{enc}_{2n}}\right) := \mathsf{KeyEncode}(k)$$

such that:

$$\mathsf{KeyEncode} : \mathbb{B} -\!\!> ; \text{Nibbles}^4 k \Big| -\!\!> ; \left(k_{\text{enc}_1}, \text{ellipsis}, k_{\text{enc}_{2n}}\right)(b_1, \text{ellipsis}, b_n) \Big| -\!\!> ; \left(b_1^1, b_1^2, b_2^1, b_2^2, \text{ellipsis}, b_n^1, b_n^2\right)$$

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:

$$k_{\text{enc}_i} := (b_i^1, b_i^2) := (b_i \div 16, b_i \bmod 16)$$

where $\mod$ is the remainder and $\div$ 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:

Definition 14. *Set of Nodes*

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 15. *State Trie*

The state trie is a radix-16 tree (Definition 12). 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 $k_N = k$.

Definition 16. *Branch*

A **branch** node $N_b \in \mathcal{N}_b$ is a node which has one child or more. A branch node can have at most 16 children. A **leaf** node $N_l \in \mathcal{N}_l$ is a childless node. Accordingly:

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

For each node, part of k_N is built while the trie is traversed from the root to N and another part of k_N is stored in N (Definition 17).

Definition 17. *Aggregated Prefix Key*

For any $N \in \mathcal{N}$, its key k_N is divided into an **aggregated prefix key,** $\mathrm{pk}_N^{\mathrm{Agr}}$, aggregated by Algorithm 1 and a **partial key**, pk_N of length $0 \leq l_{\mathrm{pk}_N} \leq 65535$ in nibbles such that:

$$\mathrm{pk}_N := \left(k_{\mathrm{enc}_i}, \text{ellipsis}, k_{\mathrm{enc}_{i+l_{\mathrm{pk}_N}}}\right)$$

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

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

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 (Definition 18).

Definition 18. *Index*

For $N \in \mathcal{N}_b$ and N_c child of N, we define Index_N function as:

$$\mathsf{Index}_N : \{N_C \in \mathcal{N} | N_c \text{is a child of} N\} - \& > ; \mathrm{Nibbles}_1^4 N_c - \& > ; i$$

such that

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

Algorithm 1 Aggregate-Key

Require: $P_N := (\text{TrieRoot} = N_1, \ldots, N_j = N)$

$pk_N^{Agr} \leftarrow \phi$
$i \leftarrow 1$
for all $N_i \in P_N$ **do**
    $pk_N^{Agr} \leftarrow pk_N^{Agr} \| pk_{N_i} \| \text{Index}_{N_i}(N_{i+1})$
end for
$pk_N^{Agr} \leftarrow pk_N^{Agr} \| pk_{N_i}$
return pk_N^{Agr}

Algorithm 1. Aggregate-Key

Assuming that P_N is the path (Definition 2) from the trie root to node N, Algorithm 1 rigorously demonstrates how to build pk_N^{Agr} while traversing P_N.

Definition 19. Node Value

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

$$\text{Node Header} \| \text{Partial Key} \| \text{Node Subvalue}$$

Formally noted as:

$$v_N := \text{Head}_N \| \text{Enc}_{\text{HE}}(pk_N) \| sv_N$$

where

- Head_N is the node header from Definition 20
- pk_N is the partial key from Definition 17
- Enc_{HE} is hex encoding (Definition 172)
- sv_N is the node subvalue from Definition 22

Definition 20. Node Header

The **node header**, consisting of ≥ 1 bytes, $N_1 \text{ellipsis } N_n$, specifies the node variant and the partial key length (Definition 17). Both pieces of information can be represented in bits within a single byte, N_1, where the amount of bits of the variant, v, and the bits of the partial key length, p_l varies.

$$v = \begin{cases} 01 & \text{Leaf} & p_l = 2^6 \\ 10 & \text{Branch Node with} k_N \notin \mathcal{K} & p_l = 2^6 \\ 11 & \text{Branch Node with} k_N \in \mathcal{K} & p_l = 2^6 \\ 001 & \text{Leaf containing a hashed subvalue} & p_l = 2^5 \\ 0001 & \text{Branch containing a hashed subvalue} & p_l = 2^4 \\ 00000000 & \text{Empty} & p_l = 0 \\ 00010000 & \text{Reserved for compact encoding} & \end{cases}$$

If the value of p_l is equal to the maximum possible value the bits can hold, such as 63 ($2^6 - 1$) in case of the 01 variant, then the value of the next 8 bits (N_2) are added the the length. This process is repeated for every N_n where $N_n = 2^8 - 1$. Any value smaller than the maximum possible value of N_n implies that the next value of N_{n+1} should not be added to the length.

The hashed subvalue for variants 001 and 0001 is described in Definition 23. The variant 0001 can be distinguished from 00010000 due to the fact that the following 4 bits of the first variant never equal zero.

Formally, the length of the partial key, pk_N^l, is defined as:

$$\text{pk}_N^l = p_l + N_n + N_{n+x} + \text{ellipsis} + N_{n+x+y}$$

as long as $p_l = m$, $N_{n+x} = 2^8 - 1$ and $N_{n+x+y} < 2^8 - 1$, where m is the maximum possible value that p_l can hold.

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 dependency is encompassed into the subvalue part of the node value which recursively depends on the Merkle value of its children. Additionally, as Section 2.2.1 clarifies, the Merkle proof of each **child trie** must be updated first before the final Polkadot state root can be calculated.

We use the auxiliary function introduced in Definition 21 to encode and decode information stored in a branch node.

Definition 21. *Children Bitmap*

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

$$\text{ChildrenBitmap}: \mathcal{N}_b - \&> ; \mathbb{B}_2 N - \&> ; (b_{15}, \text{ellipsis}, b_8, b_7, \text{ellipsis}, b_0)_2$$

where

$$b_i := \{\big(1, \exists\, N_c \in \mathcal{N} : k_{N_c} = k_{N_b} \| i \| pk_{N_c}\big), (0, \text{otherwise}).$$

Definition 22. *Subvalue*

For a given node N, the **subvalue** of N, formally referred to as sv_N, is determined as follows:

$$sv_N := \{(\text{StoredValue}_{\text{SC}}), (\text{Enc}_{\text{SC}}(\text{ChildrenBitmap}(N) \| \text{StoredValue}_{\text{SC}} | \text{Enc}_{\text{SC}}\big(H\big(N_{C_1}\big)\big), \text{ellipsis}, \text{Enc}_{\text{SC}}\big(H\big(N_{C_n}\big)\big)\big).$$

where the first variant is a leaf node and the second variant is a branch node.

$$\text{StoredValue}_{\text{SC}} := \begin{cases} \text{Enc}_{\text{SC}}(\text{StoredValue}(k_N)) & \text{if StoredValue}(k_N) = v \\ \phi & \text{if StoredValue}(k_N) = \phi \end{cases}$$

$N_{C_1} \text{ellipsis}\, N_{C_n}$ with $n \leq 16$ are the children nodes of the branch node N.

- Enc_{SC} is defined in Section 11.2.
- StoredValue, where v can be empty, is defined in Definition 11.
- H is defined in Definition 24.
- $\text{ChildrenBitmap}(N)$ is defined in Definition 21.

The trie deviates from a traditional Merkle tree in that the node value (Definition 19), v_N, is presented instead of its hash if it occupies less space than its hash.

Definition 23. *Hashed Subvalue*

To increase performance, a merkle proof can be generated by inserting the hash of a value into the trie rather than the value itself (which can be quite large). If merkle proof computation with node hashing is explicitly executed via the Host API (Section 15.1.8.2), then any value larger than 32 bytes is hashed, resulting in that hash being used as the subvalue (Definition 22) under the corresponding key. The node header must specify the variant 001 and 0001 respectively for leaves containing a hash as their subvalue and for branches containing a hash as their subvalue (Definition 20).

Definition 24. Merkle Value

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

$$H : \mathbb{B} - \& > ; U_{i-\& > ; 0}^{32} \mathbb{B}_{32} H(N) : \{(v_N, \|v_N\| \& < ; 32 \text{and} N \neq R), (\text{Blake2b}(v_n), \|v_N\| \& > ; = 32 \text{or} N = R).$$

Where v_N is the node value of N (Definition 19) and R is the root of the trie. The **Merkle hash** of the trie is defined to be $H(R)$.

2.2. Child Storage

As clarified in Section 2.1, the Polkadot state storage implements a hash table for inserting and reading key-value entries. The child storage works the same way but is stored in a separate and isolated environment. Entries in the child storage are not directly accessible via querying the main state storage.

The Polkadot Host supports as many child storages as required by Runtime and identifies each separate child storage by its unique identifying key. Child storages are usually used in situations where Runtime deals with multiple instances of a certain type of objects such as Parachains or Smart Contracts. In such cases, the execution of the Runtime entry might result in generating repeated keys across multiple instances of certain objects. Even with repeated keys, all such instances of key-value pairs must be able to be stored within the Polkadot state.

In these situations, the child storage can be used to provide the isolation necessary to prevent any undesired interference between the state of separated instances. The Polkadot Host makes no assumptions about how child storages are used, but provides the functionality for it via the Host API (Chapter 16).

2.2.1. Child Tries

The child trie specification is the same as the one described in Section 2.1.3. Child tries have their own isolated environment. Nonetheless, the main Polkadot state trie depends on them by storing a node (K_N, V_N) which corresponds to an individual child trie. Here, K_N is the child storage key associated to the child trie, and V_N is the Merkle value of its corresponding child trie computed according to the procedure described in Section 2.1.4.

The Polkadot Host API (Chapter 16) allows the Runtime to provide the key K_N in order to identify the child trie, followed by a second key in order to identify the value within that child trie. Every time a child trie is modified, the Merkle proof V_N of the child trie stored in the Polkadot state must be updated first. After that, the final Merkle proof of the Polkadot state can be computed. This mechanism provides a proof of the full Polkadot state including all its child states.

Chapter 3. State Transition

Like any transaction-based transition system, Polkadot's state is changed by 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 a Runtime (Definition 1). For easy upgradability this Runtime is presented as a Wasm blob. Nonetheless, the Polkadot Host needs to be in constant interaction with the Runtime (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.

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

3.1. Interacting with the Runtime

The Runtime (Definition 1) is the code implementing the logic of the chain. This code is decoupled from the Polkadot Host to make the the logic of the chain easily upgradable without the need to upgrade the Polkadot Host itself. The general procedure to interact with the Runtime is described by Algorithm 2.

Algorithm 2 Interact-With-Runtime

Require: $F, H_b(B), (A_1, \ldots, A_n)$

1: $S_B \leftarrow$ Set-State-At$(H_b(B))$
2: $A \leftarrow Enc_{SC}((A_1, \ldots, A_n))$
3: Call-Runtime-Entry$(R_B, RE_B, F, A, A_{len})$

Algorithm 2. Interact-With-Runtime

where

- F is the runtime entry call.
- $H_b(B)$ is the block hash indicating the state at the end of B.
- A_1, ellipsis , A_n are arguments to be passed to the runtime entry.

In this section, we describe the details upon which the Polkadot Host is interacting with the Runtime. In particular, Set-State-At and Call-Runtime-Entry procedures called by Algorithm 2 are explained in Definition 26 and Definition 32 respectively. R_B is the Runtime code loaded from S_B, as described in Definition 25, and RE_B is the Polkadot Host API, as described in Definition 174.

3.1.1. Loading the Runtime Code

The Polkadot Host 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 ASCII byte representation of the string `:code` (Chapter 12). As a result of storing the Runtime as part of the state, the Runtime code itself becomes state sensitive and calls to Runtime can change the Runtime code itself. Therefore the Polkadot Host needs to always make sure to provide the Runtime corresponding to the state in which the entry has been called. Accordingly, we define R_B (Definition 25).

The initial Runtime code of the chain is provided as part of the genesis state (Chapter 12) and subsequent calls to the Runtime have the ability to, in turn, upgrade the Runtime by replacing this Wasm blob with the help of the storage API (Chapter 15).

Definition 25. Runtime Code at State

By R_B, we refer to the Runtime code stored in the state storage at the end of the execution of block B.

3.1.2. Code Executor

The Polkadot Host executes the calls of Runtime entries inside a Wasm Virtual Machine (VM), which in turn provides the Runtime with access to the Polkadot Host API. This part of the Polkadot Host is referred to as the *Executor*.

Definition 26 introduces the notation for calling the runtime entry which is used whenever an algorithm of the Polkadot Host needs to access the runtime.

It is acceptable behavior that the Runtime panics during execution of a function in order to indicate an error. The Polkadot Host must be able to catch that panic and recover from it.

In this section, we specify the general setup for an Executor that calls into the Runtime. In Runtime API we specify the parameters and return values for each Runtime entry separately.

Definition 26. Call Runtime Entry

By

$$\text{Call-Runtime-Entry}(R, RE, \text{Runtime-Entry}, A, A_{\leq n})$$

we refer to the task using the executor to invoke the while passing an A_1, ellipsis , A_n argument to it and using the encoding described in Section 3.1.2.2.

3.1.2.1. Memory Management

The Polkadot Host is responsible for managing the WASM heap memory starting at the exported symbol as a part of implementing the allocator Host API (Chapter 22) and the same allocator should be used for any other heap allocation to be used by the Polkadot Runtime.

The size of the provided WASM memory should be based on the value of the storage key (an unsigned 64-bit integer), where each page has the size of 64KB. This memory shoule be made available to the Polkadot Runtime for import under the symbol name `memory`.

3.1.2.2. Sending Data to a Runtime Entry

In general, all data exchanged between the Polkadot Host and the Runtime is encoded using SCALE codec described in Section 11.2. Therefore all runtime entries have the following identical Wasm function 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 SCALE encoded into a byte array B (Section 11.2) and copied into a section of Wasm shared memory managed by the shared allocator described in Section 3.1.2.1.

When the Wasm method , corresponding to the entry, is invoked, two integers are passed as arguments. The first argument is set to the memory address of the byte array B in Wasm memory. The second argument sets the length of the encoded data stored in B.

3.1.2.3. Receiving Data from a Runtime Entry

The value which is returned from the invocation is an integer, representing two consecutive 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.

3.1.2.4. Handling Runtimes update to the State

In order for the Runtime to carry on various tasks, it manipulates the current state by means of executing calls to various Polkadot Host APIs (Host API). It is the duty of Host APIs to determine the context in which these changes should persist. For example, if Polkdot Host needs to validate a transaction using entry (Section 26.4.1), it needs to sandbox the changes to the state just for that Runtime call and prevent the global state of the system from being influence by the call to such a Runtime entry. This includes reverting the state of function calls which return errors or panic.

As a rule of thumb, any state changes resulting from Runtime entries are not persistent with the exception of state changes resulting from calling `Core_execute_block` (Section 26.1.2) while Polkadot Host is importing a block (Section 3.3.2).

For more information on managing multiple variant of state see Section 3.3.3.

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 state transitions. This section describes extrinsics and their inclusion into blocks.

3.2.1. Preliminaries

The extrinsics are divided into two main categories defined as follows:

Transaction extrinsics are extrinsics which are signed using either of the key types (Section 10.5)

and broadcasted between the nodes. **Inherent extrinsics** are unsigned extrinsics which are generated by Polkadot Host 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.

The Polkadot Host does not specify or limit the internals of each extrinsics and those are defined and dealt with by the Runtime (Definition 1). From the Polkadot Host point of view, each extrinsics is simply a SCALE-encoded blob (Section 11.2).

3.2.2. Transactions

Transaction are submitted and exchanged through *Transactions* network messages (Section 4.8.5). Upon receiving a Transactions message, the Polkadot Host decodes the SCALE-encoded blob and splits it into individually SCALE-encoded transactions.

Alternative transaction can be submitted to the host by offchain worker through the Host API (Section 19.1.2).

Any new transaction should be submitted to the Runtime (Section 26.4.1). This will allow the Polkadot Host to check the validity of the received transaction against the current stat and if it should be gossiped to other peers. If considers the submitted transaction as valid, the Polkadot Host should store it for inclusion in future blocks. The whole process of handeling new transactions is described in more detail by Algorithm 3.

Additionally valid transactions that are supposed to be gossiped are propagated to connected peers of the Polkadot Host. While doing so the Polkadot Host should keep track of peers already aware of each transaction. This includes peers which have already gossiped the transaction to the node as well as those to whom the transaction has already been sent. This behavior is mandated to avoid resending duplicates and unnecessarily overloading the network. To that aim, the Polkadot Host should keep a *transaction pool* and a *transaction queue* defined as follows:

Definition 27. Transaction Queue

> 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 (Section 4.8.5). The **Transaction Pool**, formally referred to as TP, is a hash table in which the Polkadot Host keeps the list of all valid transactions not in the transaction queue.

Furthermore Algorithm 3 updates the transaction pool and the transaction queue according to the received message:

Algorithm 3 Validate-Transactions-and-Store

```
L ← Dec_SC(M_T)
for all {T ∈ L ∣ T ∉ TQ ∧ T ∉ TP } do
    B_d ← Head(Longest-Chain(BT ))
    N ← H_n(B_d)
    R ← Call-Runtime-Entry(TaggedTransactionQueue_validate_transaction , N, T )
    if Valid(R) then
        if Requires(R) ∩ {T ∈ (TQ ∪ B_d)} ⊆ Provided-Tags(T ) then
            Insert-At(TQ , T , Requires(R), Priority(R))
        else
            Add-To(TP , T )
        end if
        Maintain-Transaction-Pool()
        if ShouldPropagate(R) then
            Propagate(T )
        end if
    end if
end for
```

Algorithm 3. Validate-Transactions-and-Store

where

- M_T is the transaction message (offchain transactions?)
- Dec_{SC} decodes the SCALE encoded message.
- Longest-Chain is defined in Definition 8.
- `TaggedTransactionQueue_validate_transaction` is a Runtime entry specified in Section 26.4.1 and $Requires(R)$, $Priity(R)$ 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. The Polkadot Host 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 4.
- ShouldPropagate indictes whether the transaction should be propagated based on the `Propagate` field in the `ValidTransaction`` type as defined in Definition 197, which is returned by `TaggedTransactionQueue_validate_transaction`.
- Propagate(T) sends T to all connected peers of the Polkadot Host who are not already aware of T.

Algorithm 4 Maintain-Transaction-Pool

```
Scan the pool for ready transactions
Move them to the transaction queue
Drop invalid transactions
```

Algorithm 4. Maintain-Transaction-Pool

This has not been defined yet.

3.2.3. Inherents

Inherents are unsigned extrinsic inserted into a block by the block author and as a result are not stored in the transaction pool or gossiped across the network. Instead they are generated by the Polkadot Host by passing the required inherent data, as listed in Table 1), to the Runtime method `BlockBuilder_inherent_extrinsics` (Section 26.3.3). The then returned extrinsics should be included in the current block as explained in Algorithm 12.

Table 1. Inherent Data

Identifier	Value Type	Description
timstap0	Unsigned 64-bit integer	Unix epoch time (Definition 4)
uncles00	Array of block headers	Provides a list of potential uncle block headers (Definition 28) for a given block

3.2.3.1. Definition: Inherent Data

`Inherent-Data` is a hashtable (Definition 166), an array of key-value pairs consisting of the inherent 8-byte identifier and its value, representing the totality of inherent extrinsics included in each block. The entries of this hash table which are listed in Table 1 are collected or generated by the Polkadot Host and then handed to the Runtime for inclusion (Algorithm 12).

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 (Section 3.3.1). Like any other replicated state machines, state inconsistency can occure between Polkadot replicas. Section 3.3.3 gives an overview of how a Polkadot Host node manages multiple variants of the state.

3.3.1. Block Format

A Polkadot block consists a *block header* (Definition 28) and a *block body* (Definition 31). The *block body* in turn is made up out of *extrinsics* , which represent the generalization of the concept of *transactions*. *Extrinsics* can contain any set of external data the underlying chain wishes to validate and track.

Definition 28. Block Header

The **header of block B**, $H_h(B)$, is a 5-tuple containing the following elements:

- **parent_hash:** formally indicated as H_p, is the 32-byte Blake2b hash (Section 10.2) of the SCALE encoded parent block header (Definition 30).
- **number:** formally indicated as 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 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 the Polkadot Host. This element is formally referred to as 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 H_d (Definition 29).

Definition 29. Header Digest

The header **digest** of block B formally referred to by $H_d(B)$ is an array of **digest items** H_d^i's, known as digest items of varying data type (Definition 162) such that:

$$H_d(B) := H_d^1, \text{ellipsis}, H_d^n$$

where each digest item can hold one of the following type identifiers:

$$H_d^n = \begin{cases} 0 & m \\ 4 & (E_{\text{id}}, \text{CM}) \\ 5 & (E_{\text{id}}, S_B) \\ 6 & (E_{\text{id}}, P) \\ 8 & \phi \end{cases}$$

Where E_{id} is the unique consensus engine identifier (Definition 64) and $\mathbb{B}$:

- Id *0*, the **Non-System** digest, which is opaque to the native code. m is an opaque byte array.
- Id *4*, the **Consensus Message**, contains messages from the Runtime to the consensus engine (Section 5.1.2).
- Id *5*, the **Seal**, is the data produced by the consensus engine and proving the authorship of the block producer. Respectively, S_B is the signature (Definition 78) of the block producer. In particular, the Seal digest item must be the last item in the digest array and must be stripped off by the Polkadot Host before the block is submitted to any Runtime function including for validation. The Seal must be added back to the digest afterward.
- Id *6*, the **Pre-Runtime** digest, contains the BABE Pre-Digest item (Definition 77).
- Id *8*, the **Runtime Environment Updated** digest, indicates that changes regarding the Runtime code or heap pages (Section 3.1.2.1) occurred. No additional data is provided.

Definition 30. Header Hash

The **block header hash of block** B, $H_h(B)$, is the hash of the header of block B encoded by simple codec:

$$H_h(B) := \text{Blake2b}(\text{Enc}_{SC}(\text{Head}(B)))$$

3.3.1.1. Justified Block Header

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

- **block_header** the complete block header (Definition 28) and denoted by $\text{Head}(B)$.
- **justification**: as defined by the consensus specification indicated by $\text{Just}(B)$ as defined in Definition 86.
- **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 256-bit.

Definition 31. Block Body

The block body consists of an sequence of extrinsics, each encoded as a byte array. The content of an extrinsic is completely opaque to the Polkadot Host. As such, from the point of the Polkadot Host, and is simply a SCALE encoded array of byte arrays. The **body of Block** B represented as $\text{Body}(B)$ is defined to be:

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

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

3.3.2. Importing and Validating Block

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

New blocks can be received by the Polkadot Host via other peers (Section 4.8.2) or from the Host's own consensus engine (Section 5.2). Both the Runtime and the Polkadot Host then need to work together to assure block validity. A block is deemed valid if the block author had authorship rights for the slot in which the block was produce as well as if the transactions in the block constitute a valid transition of states. The former criterion is validated by the Polkadot Host according to the block production consensus protocol. The latter can be verified by the Polkadot Host invoking entry into the Runtime as (Section 26.1.2) as a part of the validation process. Any state changes created by this function on successful execution are persisted.

The Polkadot Host implements Algorithm 5 to assure the validity of the block.

Algorithm 5 Import-and-Validate-Block

```
Require: B, Just(B)
Set-Storage-State-At(P(B))
if Just(B) ∈ □ then
    Verify-Block-Justification(B, Just(B))
    if B is Finalized and P(B) is not Finalized then
        Mark-as-Final(P(B))
    end if
end if
if H_p(B) ∉ PBT then
    return
end if
Verify-Authorship-Right(Head(B))
B ← Remove-Seal(B)
R ← Call-Runtime-Entry(Core_execute_block, B)
B ← Add-Seal(B)
if R = True then
    Persist-State()
end if
```

Algorithm 5. Import-and-Validate-Block

where

- Remove-Seal removes the Seal digest from the block (Definition 29) before submitting it to the Runtime.
- Add-Seal adds the Seal digest back to the block (Definition 29) for later propagation.
- Persist-State implies the persistence of any state changes created by `Core_execute_block` (Section 26.1.2) on successful execution.
- PBT is the pruned block tree (Definition 5).
- Verify-Authorship-Right is part of the block production consensus protocol and is described in Algorithm 10.
- *Finalized block* and *finality* are defined in Section 5.3.

3.3.3. Managaing Multiple Variants of State

Unless a node is committed to only update its state according to the finalized block (Definition 95), 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 (Section 2.1.3) facilitates and optimizes storing and switching between multiple variants of the state storage, the Polkadot Host does not specify how a node is required to accomplish this task. Instead, the Polkadot Host is required to implement Set-State-At (Definition 32):

Definition 32. Set State At Block

The function:

$$\text{Set-State-At}(B)$$

in which B is a block in the block tree (Definition 5), 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.

3.3.4. Changes Trie

Changes Tries are still work-in-progress and are currently **not** used in Polkadot. Additionally, the implementation of Changes Tries might change considerably.

Polkadot focuses on light client friendliness and therefore implements functionalities that allows identifying changes in the state of the blockchain without the requirement to search through the entire chain. The **changes trie** is a radix-16 tree data structure (Definition 12) and maintained by the Polkadot Host. It stores different types of storage changes made by each individual block separately.

The primary method for generating the changes trie is provided to the Runtime with the Host API (Section 15.1.9). The Runtime calls that function shortly before finalizing the block, the Polkadot Host must then generate the changes trie based on the storage changes which occurred during block production or execution. In order to provide this API function, it is imperative that the Polkadot Host implements a mechanism to keep track of the changes created by individual blocks, as mentioned in Section 2.1 and Section 3.3.3. The changes trie stores three different types of changes.

The changes trie itself is not part of the block, but a separately maintained database by the Polkadot Host. The Merkle proof of the changes trie must be included in the block digest (Definition 29) and gets calculated as described in Section 2.1.4. The root calculation only considers pairs which were generated on the individual block and does not consider pairs which were generated at previous blocks.

This separately maintained database by the Polkadot Host is intended to be used by "proof servers", where its implementation and behavior has not been fully defined yet. This is considered future-reserved

As clarified in the individual sections of each type, not all of those types get generated on every block. But if conditions apply, all those different types of pairs get inserted into the same changes trie, therefore only one changes trie Root gets generated for each block.

Definition 33. Inserted Key-Value Pairs

The **inserted key-value pair stored in the nodes of changes trie** is formally defined as:

$$(K_C, V_C)$$

Where K_C is a SCALE-encoded tuple:

$$\text{Enc}_{SC}\left(\text{Type}_{V_C}, H_i(B_i), K\right)$$

and

$$V_C = \text{Enc}_{SC}(C_{value})$$

is a SCALE encoded byte array.

Furthermore, K represents the changed storage key, $H_i(B_i)$ refers to the block number at which this key is inserted into the changes trie (Definition 28) and Type_{V_C} is an index defining the type C_{value} according to:

$$C_{value} = \begin{cases} 1 & (e_i, \text{ellipsis}, e_k) \\ 2 & (H_i(B_k), \text{ellipsis}, H_i(B_m)) \\ 3 & H_r(\text{Child-Changes-Trie}) \end{cases}$$

where:

- *1* is a list of extrinsics indices and e_n refers to the index of the extrinsic within the block.
- *2* is a list of block numbers.
- *3* is the child changes trie.

3.3.4.1. Key to extrinsics pairs

This key-value pair stores changes which occurred in an individual block. Its value is a SCALE encoded array containing the indices of the extrinsics that caused any changes to the specified key. The key-value pair is defined as (clarified in Section 3.3.4):

$$(1, H_i(B_i), K) - \& > ; (e_i, \text{ellipsis}, e_k)$$

The indices are unsigned 32-bit integers and their values are based on the order in which each extrinsics appears in the block (indexing starts at 0). The Polkadot Host generates those pairs for every changed key on each and every block. Child storages have their own changes trie (Section 3.3.4.3).

3.3.4.2. Key to block pairs

This key-value pair stores changes which occurred in a certain range of blocks. Its value is a SCALE encoded array containing block numbers in which extrinsics caused any changes to the specified key. The key-value pair is defined as (clarified in section Section 3.3.4):

$$(2, H_i(B_i), K) - \& > ; (H_i(B_k), \text{ellipsis}, H_i(B_m))$$

The block numbers are represented as unsigned 32-bit integers. There are multiple "levels" of those

pairs, and the Polkadot Host does **not** generate those pairs on every block. The genesis state contains the key `:changes_trie` where its unsigned 64-bit value is a tuple of two 32-bit integers:

- **interval** - The interval (in blocks) at which those pairs should be created. If this value is less or equal to 1 it means that those pairs are not created at all.
- **levels** - The maximum number of "levels" in the hierarchy. If this value is 0 it means that those pairs are not created at all.

For each level from 1 to *levels*, the Polkadot Host creates those pairs on every -nth block.

For example, let's say *interval* is set at and is set at . This means there are now three levels which get generated at three different occurrences:

1. **Level 1** - Those pairs are generated at every 4^1-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. This level only considers block numbers of the last four (4^1) blocks.
    - Example: this level occurs at block 4, 8, 12, 16, 32, etc.
2. **Level 2** - Those pairs are generated at every 4^2-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. This level only considers block numbers of the last 16 (4^2) blocks.
    - Example: this level occurs at block 16, 32, 64, 128, 256, etc.
3. **Level 3** - Those pairs are generated at every 4^3-nth block, where the pair value contains the block numbers of every block that changed the specified storage key. this level only considers block number of the last 64 (4^3) blocks.
    - Example: this level occurs at block 64, 128, 196, 256, 320, etc.

3.3.4.3. Key to Child Changes Trie pairs

The Polkadot Host generates a separate changes trie for each child storage, using the same behavior and implementation as describe in Section 3.3.4.1. Additionally, the changed child storage key gets inserted into the primary, non-Child changes trie where its value is a SCALE encoded byte array containing the Merkle root of the Child changes trie. The key-value pair is defined as:

$$(3, H_i(B_i), K) - \& > ; H_r(\text{Child-Changes-Trie})$$

The Polkadot Host creates those pairs for every changes child key for each and every block.

Chapter 4. Networking

This chapter in its current form is incomplete and considered work in progress. Authors appreciate receiving request for clarification or any reports regarding deviation from the current Polkadot network protocol. This can be done through filing an issue in Polkadot Specification repository.

4.1. Introduction

The Polkadot network is decentralized and does not rely on any central authority or entity for achieving its fullest potential of provided functionality. The networking protocol is based on a family of open protocols, including protocol implemented *libp2p* e.g. the distributed Kademlia hash table which is used for peer discovery.

This chapter walks through the behavior of the networking implementation of the Polkadot Host and defines the network messages. The implementation details of the *libp2p* protocols used are specified in external sources as described in Section 4.2

4.2. External Documentation

Complete specification of the Polkadot networking protocol relies on the following external protocols:

- libp2p - *libp2p* is a modular peer-to-peer networking stack composed of many modules and different parts. includes the multiplexing protocols and .
- libp2p addressing - The Polkadot Host uses the *libp2p* addressing system to identify and connect to peers.
- Kademlia - *Kademlia* is a distributed hash table for decentralized peer-to-peer networks. The Polkadot Host uses Kademlia for peer discovery.
- Noise - The *Noise* protocol is a framework for building cryptographic protocols. The Polkadot Host uses Noise to establish the encryption layer to remote peers.
- mplex - *mplex* is a multiplexing protocol developed by *libp2p*. The protocol allows dividing a connection to a peer into multiple substreams, each substream serving a specific purpose. Generally, Polkadot Host implementers are encouraged to prioritize implementing , since it is the de-facto standard in Polkadot.*mplex* is only required to communicate with js-lip2p.
- yamux - *yamux* is a multiplexing protocol like *mplex* and developed by HashiCorp. It is the de-facto standard for the Polkadot Host. This protocol should be prioritized over *mplex*. Section 4.7 describes the subprotocol in more detail.
- Protocol Buffers - Protocol Buffers is a language-neutral, platform-neutral mechanism for serializing structured data and is developed by Google. The Polkadot Host uses Protocol Buffers to serialize specific messages, as clarified in Section 4.8.

4.3. Node Identities

Each Polkadot Host node maintains an ED25519 key pair which is used to identify the node. The

public key is shared with the rest of the network allowing the nodes to establish secure communication channels.

Each node must have its own unique ED25519 key pair. When two or more nodes use the same key, the network will interpret those nodes as a single node, which will result in undefined behavior and can result in equivocation. Furthermore, the node's *PeerId* as defined in Definition 34 is derived from its public key. *PeerId* is used to identify each node when they are discovered in the course of the discovery mechanism described in Section 4.4.

Definition 34. PeerId

The Polkadot node's *PeerId*, formally referred to as P_{id}, is derived from the ED25519 public key and is structured as defined in the libp2p specification.

4.4. Discovery mechanism

The Polkadot Host uses various mechanisms to find peers within the network, to establish and maintain a list of peers and to share that list with other peers from the network as follows:

- **Bootstrap nodes** are hard-coded node identities and addresses provided by the genesis state (Chapter 12).
- **mDNS** is a protocol that performs a broadcast to the local network. Nodes that might be listening can respond to the broadcast. The libp2p mDNS specification defines this process in more detail. This protocol is an optional implementation detail for Polkadot Host implementers and is not required to participate in the Polkadot network.
- **Kademlia requests** invoking Kademlia requests, where nodes respond with their list of available peers. Kademlia requests are performed on a specific substream as described in Section 4.7.

4.5. Connection establishment

Polkadot nodes connect to peers by establishing a TCP connection. Once established, the node initiates a handshake with the remote peers on the encryption layer. An additional layer on top of the encryption layer, known as the multiplexing layer, allows a connection to be split into substreams, as described by the yamux specification, either by the local or remote node.

The Polkadot node supports two types of substream protocols. Section 4.7 describes the usage of each type in more detail:

- **Request-Response substreams**: After the protocol is negotiated by the multiplexing layer, the initiator sends a single message containing a request. The responder then sends a response, after which the substream is then immediately closed. The requests and responses are prefixed with their LEB128 encoded length.
- **Notification substreams**. After the protocol is negotiated, the initiator sends a single handshake message. The responder can then either accept the substream by sending its own handshake or reject it by closing the substream. After the substream has been accepted, the

initiator can send an unbound number of individual messages. The responder keeps its sending side of the substream open, despite not sending anything anymore, and can later close it in order to signal to the initiator that it no longer wishes to communicate.

Handshakes and messages are prefixed with their LEB128 encoded lengths. A handshake can be empty, in which case the length prefix would be *0*.

Connections are established by using the following protocols:

- `/noise` - a protocol that is announced when a connection to a peer is established.
- `/multistream/1.0.0` - a protocol that is announced when negotiating an encryption protocol or a substream.
- `/yamux/1.0.0` - a protocol used during the *mplex* or *yamux* negotiation. See Section 4.7 for more information.

The Polkadot Host can establish a connection with any peer of which it knows the address. The Polkadot Host supports multiple networking protocols:

- **TCP/IP** with addresses in the form of `/ip4/1.2.3.4/tcp/` to establish a TCP connection and negotiate encryption and a multiplexing layer.
- **Websockets** with addresses in the form of `/ip4/1.2.3.4/ws/` to establish a TCP connection and negotiate the Websocket protocol within the connection. Additionally, encryption and multiplexing layer is negotiated within the WebSocket connection.
- **DNS** addresses in form of `/dns/example.com/tcp/` and `/dns/example.com/ws/`.

The addressing system is described in the libp2p addressing specification. After a base-layer protocol is established, the Polkadot Host will apply the Noise protocol to establish the encryption layer as described in Section 4.6.

4.6. Encryption Layer

Polkadot protocol uses the *libp2p* Noise framework to build an encryption protocol. The Noise protocol is a framework for building encryption protocols. *libp2p* utilizes that protocol for establishing encrypted communication channels. Refer to the libp2p Secure Channel Handshake specification for a detailed description.

Polkadot nodes use the XX handshake pattern to establish a connection between peers. The three following steps are required to complete the handshake process:

1. The initiator generates a keypair and sends the public key to the responder. The Noise specification and the libp2p PeerId specification describe keypairs in more detail.
2. The responder generates its own key pair and sends its public key back to the initiator. After that, the responder derives a shared secret and uses it to encrypt all further communication. The responder now sends its static Noise public key (which may change anytime and does not need to be persisted on disk), its *libp2p* public key and a signature of the static Noise public key signed with the *libp2p* public key.
3. The initiator derives a shared secret and uses it to encrypt all further communication. It also

sends its static Noise public key, *libp2p* public key and signature to the responder.

After these three steps, both the initiator and responder derive a new shared secret using the static and session-defined Noise keys, which are used to encrypt all further communication.

4.7. Protocols and Substreams

After the node establishes a connection with a peer, the use of multiplexing allows the Polkadot Host to open substreams. *libp2p* uses the *mplex protocol* or the *yamux protocol* to manage substreams and to allow the negotiation of *application-specific protocols*, where each protocol serves a specific utility.

The Polkadot Host uses multiple substreams whose usage depends on a specific purpose. Each substream is either a *Request-Response substream* or a *Notification substream*, as described in Section 4.5.

The prefixes on those substreams are known as protocol identifiers and are used to segregate communications to specific networks. This prevents any interference with other networks. `dot` is used exclusively for Polkadot. Kusama, for example, uses the protocol identifier `ksmcc3`.

- `/ipfs/ping/` - Open a standardized substream *libp2p* to a peer and initialize a ping to verify if a connection is still alive. If the peer does not respond, the connection is dropped. This is a *Request-Response substream*.

  Further specification and reference implementation are available in the libp2p documentation.

- `/ipfs/id/1.0.0` - Open a standardized *libp2p* substream to a peer to ask for information about that peer. This is a *Request-Response substream*.

  Further specification and reference implementation are available in the libp2p documentation.

- `/dot/kad` - Open a standardized substream for Kademlia `FIND_NODE` requests. This is a *Request-Response substream*, as defined by the *libp2p* standard.

  Further specification and reference implementation are available on Wikipedia respectively the golang Github repository.

- `/dot/light/2` - a request and response protocol that allows a light client to request information about the state. This is a *Request-Response substream*.

  The messages are specified in Section 4.10.

- `/dot/block-announces/1` - a substream/notification protocol which sends blocks to connected peers. This is a *Notification substream*.

  The messages are specified in Section 4.8.1.

- `/dot/sync/2` - a request and response protocol that allows the Polkadot Host to request information about blocks. This is a *Request-Response substream*.

The messages are specified in Section 4.8.2.

- `/dot/sync/warp` - a request and response protocol that allows the Polkadot Host to perform a warp sync request. This is a *Request-Response substream*.

  The messages are specified in Section 4.9.3.

- `/dot/transactions/1` - a substream/notification protocol which sends transactions to connected peers. This is a *Notification substream*.

  The messages are specified in Section 4.8.5.

- `/dot/grandpa/1` - a substream/notification protocol that sends GRANDPA votes to connected peers. This is a *Notification substream*.

  The messages are specified in Section 4.8.6.

  For backwards compatibility reasons, `/paritytech/grandpa/1` is also a valid substream for those messages.

- `/dot/beefy/1` - a substream/notification protocol which sends signed BEEFY statements, as described in Section 5.5, to connected peers. This is a *Notification* substream.

  The messages are specified in Section 4.8.7.

  For backwards compatibility reasons, `/paritytech/beefy/1` is also a valid substream for those messages.

4.8. Network Messages

The Polkadot Host must actively communicate with the network in order to participate in the validation process or act as a full node.

The Polkadot network originally only used SCALE encoding for all message formats. Meanwhile, Protobuf has been adopted for certain messages. The encoding of each listed message is always SCALE encoded unless Protobuf is explicitly mentioned. Encoding and message formats are subject to change.

4.8.1. Announcing blocks

When the node creates or receives a new block, it must be announced to the network. Other nodes within the network will track this announcement and can request information about this block. The mechanism for tracking announcements and requesting the required data is implementation-specific.

Block announcements, requests and responses are sent over the substream as described in Definition 35.

Definition 35. Block Announce Handshake

The `BlockAnnounceHandshake` initializes a substream to a remote peer. Once established, all `BlockAnounce` messages (Definition 36) created by the node are sent to the `/dot/block-announces/1` substream.

The `BlockAnnounceHandshake` is a structure of the following format:

$$BA_h = \text{Enc}_{\text{SC}}(R, N_B, h_B, h_G)$$

where:

$$R = \begin{cases} 1 & \text{The node is a full node} \\ 2 & \text{The node is a light client} \\ 4 & \text{The node is a validator} \end{cases} \quad N_B = \text{Best block number according to the node} \quad h_B = \text{Best block hash according to the node} \quad h_G = \text{Genesis block hash according to the node}$$

Definition 36. Block Announce

The `BlockAnnounce` message is sent to the specified substream and indicates to remote peers that the node has either created or received a new block.

The message is a structure of the following format:

$$BA = \text{Enc}_{\text{SC}}(\text{Head}(B), b)$$

where:

$$\text{Head}(B) = \text{Header of the announced block} \quad b = \begin{cases} 0 & \text{Is not part of the best chain} \\ 1 & \text{Is the best block according to the node} \end{cases}$$

4.8.2. Requesting Blocks

Block requests can be used to retrieve a range of blocks from peers. Those messages are sent over the `/dot/sync/2` substream.

Definition 37. Block Request

The `BlockRequest` message is a Protobuf serialized structure of the following format:

Type	Id	Description	Value
`uint32`	1	Bits of block data to request	B_f
`oneof`		Start from this block	B_S
`bytes`	4	End at this block (*optional*)	B_e
Direction	5	Sequence direction	
`uint32`	6	Maximum amount (*optional*)	B_m

where:

- B_f indicates all the fields that should be included in the request. its **big-endian** encoded bitmask that applies to all desired fields with bitwise OR operations. For example, the B_f value to request *Header* and *Justification* is *0001 0001* (17).

Field	Value
Header	0000 0001
Body	0000 0010
Justification	0001 0000

- B_s is a Protobuf structure indicating a varying data type of the following values:

Type	Id	Description
`bytes`	2	The block hash
`bytes`	3	The block number

- B_e is either the block hash or block number depending on the value of B_s. an implementation-defined maximum is used when unspecified.
- *Direction* is a Protobuf structure indicating the sequence direction of the requested blocks. The structure is a varying data type (Definition 162) of the following format:

Id	Description
0	Enumerate in ascending order (from child to parent)
1	Enumerate in descending order (from parent to canonical child)

- B_m is the number of blocks to be returned. An implementation defined maximum is used when unspecified.

Definition 38. Block Response

The `BlockResponse` message is received after sending a `BlockRequest` message to a peer. The message is a Protobuf serialized structure of the following format:

Type	Id	Description
Repeated *BlockData*	1	Block data for the requested sequence

where *BlockData* is a Protobuf structure containing the requested blocks. Do note that the optional values are either present or absent depending on the requested fields (bitmask value). The structure has the following format:

Type	Id	Description	Value
`bytes`	1	Block header hash	Definition 30
`bytes`	2	Block header (optional)	Definition 28
repeated `bytes`	3	Block body (optional)	Definition 31
`bytes`	4	Block receipt (optional)	
`bytes`	5	Block message queue (optional)	
`bytes`	6	Justification (optional)	Definition 86
`bool`	7	Indicates whether the justification is empty (i.e. should be ignored)	

4.8.3. Requesting States

The Polkadot Host can request the state in form of a key/value list at a specified block.

When receiving state entries from the state response messages (Definition 40), the node can verify the entries with the entry proof (id *1* in *KeyValueStorage*) against the merkle root in the block header (of the block specified in Definition 39). Once the state response message claims that all entries have been sent (id *3* in *KeyValueStorage*), the node can use all collected entry proofs and validate it against the merkel root to confirm that claim.

See the the synchronization section for more information (Section 4.9).

Definition 39. State Request

A **state request** is sent to a peer to request the state at a specified block. The message is a single 32-byte Blake2 hash which indicates the block from which the sync should start.

Depending on what substream is used, he remote peer either sends back a state response (Definition 40) on the `/dot/sync/2` substream or a warp sync proof (Definition 41) on the `/dot/sync/warp`.

Definition 40. State Response

The **state response** is sent to the peer that initialized the state request (Definition 39) and contains a list of key/value entries with an associated proof. This response is sent continuously until all key/value pairs have been submitted.

Type	Id	Description
`repeated KeyValueStateEntry`	1	State entries
`bytes`	2	State proof

where *KeyValueStateEntry* is of the following format:

Type	Id	Description
`bytes`	1	Root of the entry, empty if top level
`repeated StateEntry`	2	Collection of key/values
`bool`	3	Equal 'true' if there are no more keys to return.

and *StateEntry*:

Type	Id	Description
`bytes`	1	The key of the entry
`bytes`	2	The value of the entry

4.8.4. Warp Sync

The warp sync protocols allows nodes to retrieve blocks from remote peers where authority set changes occurred. This can be used to speed up synchronization to the latest state.

See the the synchronization section for more information (Section 4.9).

Definition 41. Warp Sync Proof

The **warp sync proof** message, P, is sent to the peer that initialized the state request (Definition 39) on the `/dot/sync/warp` substream and contains accumulated proof of multiple authority set changes (Section 5.1.2). It's a datastructure of the following format:

$$P = \left(f_x \text{ellipsis } f_y, c\right)$$

$f_x \text{ellipsis } f_y$ is an array consisting of warp sync fragments of the following format:

$$f_x = \left(B_h, J^{r,\,\text{stage}}(B)\right)$$

where B_h is the last block header containing a digest item (Definition 29) signaling an authority set change from which the next authority set change can be fetched from. $J^{r,\,\text{stage}}(B)$ is the GRANDPA justification (Definition 86) and c is a boolean that indicates whether the warp sync has been completed.

4.8.5. Transactions

Transactions (Section 3.2) are sent directly to peers with which the Polkadot Host has an open transaction substream (Definition 42). Polkadot Host implementers should implement a mechanism that only sends a transaction once to each peer and avoids sending duplicates. Sending duplicate transactions might result in undefined consequences such as being blocked for bad behavior by peers.

The mechanism for managing transactions is further described in Section Section 3.2.

Definition 42. Transaction Message

The **transactions message** is the structure of how the transactions are sent over the network. It is represented by M_T and is defined as follows:

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

in which:

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

Where each E_i is a byte array and represents a separate extrinsic. The Polkadot Host is agnostic about the content of an extrinsic and treats it as a blob of data.

Transactions are sent over the `/dot/transactions/1` substream.

4.8.6. GRANDPA Messages

The exchange of GRANDPA messages is conducted on the substream. The process for the creation and distributing these messages is described in Section 5.3. The underlying messages are specified in this section.

Definition 43. *Grandpa Gossip Message*

A **GRANDPA gossip message**, M, is a varying datatype (Definition 162) which identifies the message type that is cast by a voter followed by the message itself.

$$M = \begin{cases} 0 & \text{Vote message} & V_m \\ 1 & \text{Commit message} & C_m \\ 2 & \text{Neighbor message} & N_m \\ 3 & \text{Catch-up request message} & R_m \\ 4 & \text{Catch-up message} & U_m \end{cases}$$

where:

- V_m is defined in Definition 44.
- C_m is defined in Definition 46.
- N_m is defined in Definition 47.
- R_m is defined in Definition 48.
- U_M is defined in Definition 49.

Definition 44. *GRANDPA Vote Messages*

A **GRANDPA vote message** by voter v, $M_v^{r,\,\text{stage}}$, is gossip to the network by voter v with the following structure:

$$M_v^{r,\,\text{stage}}(B) := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}}, \text{SigMsg})\text{SigMsg} := \left(\text{msg}, \text{Sig}_{v_i}^{r,\,\text{stage}}, v_{\text{id}}\right)\text{msg} := \text{Enc}_{\text{SC}}\left(\text{stage}, V_v^{r,\,\text{stage}}(B)\right)$$

where:

- r is an unsigned 64-bit integer indicating the Grandpa round number (Definition 84).
- $\text{id}_{\mathbb{V}}$ is an unsigned 64-bit integer indicating the authority Set Id (Definition 63).
- $\text{Sig}_{v_i}^{r,\,\text{stage}}$ is a 512-bit byte array containing the signature of the authority (Definition 85).
- v_{id} is a 256-bit byte array containing the *ed25519* public key of the authority.
- stage is a 8-bit integer of value *0* if it's a pre-vote sub-round, *1* if it's a pre-commit sub-round or *2* if it's a primary proposal message.
- $V_v^{r,\,\text{stage}}(B)$ is the GRANDPA vote for block B (Definition 84).

This message is the sub-component of the GRANDPA gossip message (Definition 43) of type Id 0.

Definition 45. GRANDPA Compact Justification Format

The **GRANDPA compact justification format** is an optimized data structure to store a collection of pre-commits and their signatures to be submitted as part of a commit message. Instead of storing an array of justifications, it uses the following format:

$$J^{r,\,\text{comp}}_{v_{0},\,\text{ellipsis}\,n} := \left(\left\{V^{r,\,pc}_{v_0}, \text{ellipsis}\, V^{r,\,pc}_{v_n}\right\}, \left\{\left(\text{Sig}^{r,\,pc}_{v_0}, v_{\text{id}_0}\right), \text{ellipsis}\left(\text{Sig}^{r,\,pc}_{v_n}, v_{\text{id}_n}\right)\right\}\right)$$

where:

- $V^{r,\,pc}_{v_i}$ is a 256-bit byte array containing the pre-commit vote of authority v_i (Definition 84).
- $\text{Sig}^{r,\,pc}_{v_i}$ is a 512-bit byte array containing the pre-commit signature of authority v_i (Definition 85).
- v_{id_n} is a 256-bit byte array containing the public key of authority v_i.

Definition 46. GRANDPA Commit Message

A **GRANDPA commit message** for block B in round r, $M^{r,\,\text{Fin}}_{v}(B)$, 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:

$$M^{r,\,\text{Fin}}_{v}(B) := \text{Enc}_{\text{SC}}\left(r, \text{id}_{V}, V^{r}_{v}(B), J^{r,\,\text{comp}}_{v_{0},\,\text{ellipsis}\,n}\right)$$

where:

- r is an unsigned 64-bit integer indicating the round number (Definition 84).
- id_V is the authority set Id (Definition 63).
- $V^{r}_{v}(B)$ is a 256-bit array containing the GRANDPA vote for block B (Definition 83).
- $J^{r,\,\text{comp}}_{v_{0},\,\text{ellipsis}\,n}$ is the compacted GRANDPA justification containing observed pre-commit of authorities v_0 to v_n (Definition 45).

This message is the sub-component of the GRANDPA gossip message (Definition 43) of type Id *1*.

4.8.6.1. GRANDPA Neighbor Messages

Neighbor messages are sent to all connected peers but they are not repropagated on reception. A message should be send whenever the messages values change and at least every 5 minutes. The sender should take the recipients state into account and avoid sending messages to peers that are using a different voter sets or are in a different round. Messages received from a future voter set or round can be dropped and ignored.

Definition 47. GRANDPA Neighbor Message

A **GRANDPA Neighbor Message** is defined as:

$$M^{\text{neigh}} := \text{Enc}_{\text{SC}}(v, r, \text{id}_{\mathbb{V}}, H_h(B_{\text{last}}))$$

where:

- v is an unsigned 8-bit integer indicating the version of the neighbor message, currently *1*.
- r is an unsigned 64-bit integer indicating the round number (Definition 84).
- $\text{id}_{\mathbb{V}}$ is an unsigned 64-bit integer indicating the authority Id (Definition 63).
- $H_i(B_{\text{last}})$ is an unsigned 32-bit integer indicating the block number of the last finalized block B_{last}.

This message is the sub-component of the GRANDPA gossip message (Definition 43) of type Id *2*.

4.8.6.2. GRANDPA Catch-up Messages

Whenever a Polkadot node detects that it is lagging behind the finality procedure, it needs to initiate a *catch-up* procedure. GRANDPA Neighbor messages (Definition 47) reveal the round number for the last finalized GRANDPA round which the node's peers have observed. This provides the means to identify a discrepancy in the latest finalized round number observed among the peers. If such a discrepancy is observed, the node needs to initiate the catch-up procedure explained in Section 5.4.1).

In particular, this procedure involves sending a *catch-up request* and processing *catch-up response* messages.

Definition 48. Catch-Up Request Message

A **GRANDPA catch-up request message** for round r, $M_{i,v}^{\text{Cat}-q}(\text{id}_{\mathbb{V}}, r)$, is a message sent from node i to its voting peer node v requesting the latest status of a GRANDPA round $r' > r$ of the authority set $\mathbb{V}_{\text{id}}$ along with the justification of the status and has the following structure:

$$M_{i,v}^{r,\,\text{Cat}-q} := \text{Enc}_{\text{SC}}(r, \text{id}_{\mathbb{V}})$$

This message is the sub-component of the GRANDPA Gossip message (Definition 43) of type Id *3*.

Definition 49. Catch-Up Response Message

A **GRANDPA catch-up response message** for round r, $M_{v,i}^{\text{Cat}-s}(\text{id}_V, r)$, is a message sent by a node v to node i in response of a catch-up request $M_{v,i}^{\text{Cat}-q}(\text{id}_V, r')$ in which $r \geq r'$ is the latest GRANDPA round which v has prove of its finalization and has the following structure:

$$M_{v,i}^{\text{Cat}-s} := \text{Enc}_{\text{SC}}\left(\text{id}_V, r, J_{0,\text{ellipsis}\,n}^{r,\text{pv}}(B), J_{0,\text{ellipsis}\,m}^{r,\text{pc}}(B), H_h(B'), H_i(B')\right)$$

Where B is the highest block which v believes to be finalized in round r (Definition 84). B' is the highest ancestor of all blocks voted on in the arrays of justifications $J_{0,\text{ellipsis}\,n}^{r,\text{pv}}(B)$ and $J_{0,\text{ellipsis}\,m}^{r,\text{pc}}(B)$ (Definition 86) with the exception of the equivocatory votes.

This message is the sub-component of the GRANDPA Gossip message (Definition 43) of type Id *4*.

4.8.7. GRANDPA BEEFY

The BEEFY protocol is currently in early development and subject to change.

This section defines the messages required for the GRANDPA BEEFY protocol (Section 5.5). Those messages are sent over the `/paritytech/beefy/1` substream.

Definition 50. Commitment

A **commitment**, C, contains the information extracted from the finalized block at height $H_i(B_{\text{last}})$ as specified in the message body and a datastructure of the following format:

$$C = (R_h, H_i(B_{\text{last}}), \text{id}_V)$$

where

- R_h is the MMR root of all the block header hashes leading up to the latest, finalized block.
- $H_i(B_{\text{last}})$ is the block number this commitment is for. Namely the latest, finalized block.
- id_V is the current authority set Id (Definition 81).

Definition 51. Vote Message

A **vote message**, M_v, is direct vote created by the Polkadot Host on every BEEFY round and is gossiped to its peers. The message is a datastructure of the following format:

$$M_v = \text{Enc}_{\text{SC}}\left(C, A_{\text{id}}^{\text{bfy}}, A_{\text{sig}}\right)$$

where

- C is the BEEFY commitment (Definition 50).
- $A_{\text{id}}^{\text{bfy}}$ is the ECDSA public key of the Polkadot Host.
- A_{sig} is the signature created with $A_{\text{id}}^{\text{bfy}}$ by signing the statement R_h in C.

Definition 52. Signed Commitment

A **signed commitment**, M_{sc}, is a datastructure of the following format:

$$M_{\text{SC}} = \text{Enc}_{\text{SC}}(C, S_n) S_n = \left(A_0^{\text{sig}}, \text{ellipsis } A_n^{\text{sig}}\right)$$

where

- C is the BEEFY commitment (Definition 50).
- S_n is an array where its exact size matches the number of validators in the current authority set as specified by id_V (Definition 81) in C. Individual items are of the type *Option* (Definition 164) which can contain a signature of a validator which signed the same statement (R_h in C) and is active in the current authority set. It's critical that the signatures are sorted based on their corresponding public key entry in the authority set.

  For example, the signature of the validator at index 3 in the authority set must be placed at index *3* in S_n. If not signature is available for that validator, then the *Option* variant is *None* inserted (Definition 164). This sorting allows clients to map public keys to their corresponding signatures.

Definition 53. Signed Commitment Witness

A **signed commitment witness**, M_{SC}^{w}, is a light version of the signed BEEFY commitment (Definition 52). Instead of containing the entire list of signatures, it only claims which validator signed the statement.

The message is a datastructure of the following format:

$$M_{\text{SC}}^{w} = \text{Enc}_{\text{SC}}\left(C, V_{0,\ \text{ellipsis}\ n}, R_{\text{sig}}\right)$$

where

- C is the BEEFY commitment (Definition 50).
- $V_{0,\ \text{ellipsis}\ n}$ is an array where its exact size matches the number of validators in the current authority set as specified by id_V in C. Individual items are booleans which indicate whether the validator has signed the statement (*true*) or not (*false*). It's critical that the boolean indicators are sorted based on their corresponding public key entry in the authority set.

  For example, the boolean indicator of the validator at index 3 in the authority set must be placed at index *3* in V_n. This sorting allows clients to map public keys to their corresponding boolean indicators.

- R_{sig} is the MMR root of the signatures in the original signed BEEFY commitment (Definition 52).

4.9. Synchronization

Synchronization is mostly application specific and the protocol does not mandate how synchronization must be conducted. The network messages are specified in Section 4.8. This section makes some recommendations how a synchronization mechanism can be constructed.

Many applications that interact with the Polkadot network to some extent must be able to retrieve certain information about the network. Depending on the utility, this includes validators that interact with Polkadot's consensus and need access to the full state, either from the past or just the most up-to-date state, or light clients that are only interest in the minimum information required in order to verify some claims about the state of the network, such as the balance of a specific account. To allow implemenations to quickly retrieve the required information, different types of synchronization protocols are available, respectivel Full, Fast and Warp sync suited for different needs.

4.9.1. Full Sync

The full sync protocols is the "default" protocol that's suited for many types of implementations, such as archive nodes (nodes that store everything), validators that participate in Polkadots consensus and light clients that only verify claims about the state of the network. Full sync works by listening to announced blocks (Section 4.8.1) and requesting the blocks from the announcing peers, or just the block headers in case of light clients.

The full sync protocol usually downloads the entire chain, but no such requirements must be met. If an implemenation only wants the latest, finalized state, it can combine it with protocols such as fast sync (Section 4.9.2) and/or warp sync (Section 4.9.3) to make synchronization as fast as possible.

4.9.2. Fast Sync

Fast sync works by downloading the block header history and validating the auhtority set changes (Section 5.1.1) in order to arrive at a specific (usually the most recent) header. After the desired header has been reached and verified, the state can be downloaded and imported (Section 4.8.3). Once this process has been completed, the node can proceed with a full sync.

4.9.3. Warp Sync

Warp sync (Section 4.8.4) only downloads the block headers where authority set changes occurred, so called fragments (Definition 41), and by verifying the GRANDPA justifications (Definition 86). This protocols allows nodes to arrive at the desired state much faster than fast sync.

4.10. Light Client Messages

Light clients are applications that fetch the required data that they need from a Polkadot node with an associated proof to validate the data. This makes it possible to interact with the Polkadot network without requiring to run a full node or having to trust the remote peers. The light client messages make this functionality possible.

All light client messages are protobuf encoded and are sent over the `/dot/light/2` substream.

4.10.1. Request

A message with all possible request messages. All message are sent as part of this message.

Type	Id	Description
`oneof` (`request`)		The request type

Where the `request` can be one of the following fields:

Type	Id	Description
`RemoteCallRequest`	1	A remote call request (Definition 54)
`RemoteReadRequest`	2	A remote read request (Definition 56)
`RemoteHeaderRequest`	3	A remote header request (Definition 59)
`RemoteReadChildRequest`	4	A remote read child request (Definition 58)
`RemoteChangesRequest`	5	A remote changes request (Definition 61)

4.10.2. Response

A message with all possible response messages. All message are sent as part of this message.

Type	Id	Description
`oneof` (`response`)		The response type

Where the `response` can be one of the following fields:

Type	Id	Description
`RemoteCallResponse`	1	A remote call response (Definition 55)
`RemoteReadResponse`	2	A remote read response (Definition 57)
`RemoteHeaderResponse`	3	A remote header response (Definition 60)
`RemoteChangesResponse`	4	A remote changes response (Definition 62)

4.10.3. Remote Call Messages

Execute a call to a contract at the given block.

Definition 54. Remote Call Request

Remote call request.

Type	Id	Description
`bytes`	2	Block at which to perform call
`string`	3	Method name
`bytes`	4	Call data

Definition 55. Remote Call Response

Remote call response.

Type	Id	Description
`bytes`	2	Execution proof

4.10.4. Remote Read Messages

Read a storage value at the given block.

Definition 56. Remote Read Request

Remote read request.

Type	Id	Description
`bytes`	2	Block at which to perform call
`repeated bytes`	3	Storage keys

Definition 57. Remote Read Response

Remote read response.

Type	Id	Description
`bytes`	2	Read proof

4.10.5. Remote Read Child Messages

Read a child storage value at the given block.

Definition 58. Remote Read Child Request

Remote read child request.

Type	Id	Description
bytes	2	Block at which to perform call
bytes	3	Child storage key, this is relative to the child type storage location
bytes	6	Storage keys

4.10.6. Remote Header Messages

Request a block header at the given block.

Definition 59. Remote Header Request

Remote header request.

Type	Id	Description
bytes	2	Block number to request header for

Definition 60. Remote Header Response

Remote header response.

Type	Id	Description
bytes	2	The header. Is *None* (Definition 164) if proof generation has failed (e.g. header is unknown)

4.10.7. Remote Changes Message

Remote changes messages.

Definition 61. Remote Changes Request

Remote changes request.

Type	Id	Description
`bytes`	2	Hash of the first block of the range (including first) where changes are requested
`bytes`	3	Hash of the last block of the range (including last) where changes are requested
`bytes`	4	Affected roots must be proved
`bytes`	5	Hash of the last block that we can use when querying changes
`bytes`	6	(Optional) storage child node key which changes are requested (Definition 164)
`bytes`	7	Storage key which changes are requested

Definition 62. Remote Changes Response

Remote changes response.

Type	Id	Description
`bytes`	2	Proof has been generated using block with this number as a max block.
`repeated bytes`	3	Changes proof
`repeated Pair`	4	Changes tries roots missing on the requester node
`bytes`	5	Missing changes tries roots proof.

Where `Pair` is a protobuf datastructure of the following format:

Type	Id	Description
`bytes`	1	The first element of the pair
`bytes`	2	The second element of the pair

Chapter 5. Consensus

Consensus in the Polkadot Host is achieved during the execution of two different procedures. The first procedure is the block-production and the second is finality. The Polkadot Host 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 engines 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. To verify the validity of each block, the Polkadot node must track the current list of authorities (Definition 63) for that block.

Definition 63. Authority List

The **authority list** of block B for consensus engine C noted as $\text{Auth}_C(B)$ is an array that contains the following pair of types for each of its authorities $A \in \text{Auth}_C(B)$:

$$(pk_A, w_A)$$

pk_A is the session public key (Definition 155) of authority A. And w_A is an unsigned 64-bit integer indicating the authority weight. The value of $\text{Auth}_C(B)$ is part of the Polkadot state. The value for $\text{Auth}_C(B_0)$ is set in the genesis state (Chapter 12) and can be retrieved using a runtime entry corresponding to consensus engine C.

The authorities and their corresponding weights can be retrieved from the Runtime (Section 26.7.1).

5.1.2. Runtime-to-Consensus Engine Message

The authority list (Definition 63) is part of the Polkadot state and the Runtime has the authority to update this list in the course of any state transitions. The Runtime informs the corresponding consensus engine about the changes in the authority set by adding the appropriate consensus message (Definition 64) in the form of a digest item (Definition 29) to the block header of block B which caused the transition in the authority set.

The Polkadot Host must inspect the digest header of each block and delegate consensus messages to their consensus engines. The BABE and GRANDPA consensus engine must react based on the type of consensus messages it receives. The active GRANDPA authorities can only vote for blocks that occurred after the finalized block in which they were selected. Any votes for blocks before the came into effect would get rejected.

Definition 64. Consensus Message

A **consensus message**, CM, is a digest item (Definition 29) of type *4* and consists one of the following tuple pairs, where the first item is a string which serves as an identifier.

$$\text{CM} = \begin{cases} (\text{'BABE'}, \text{CM}_b) \\ (\text{'FRNK'}, \text{CM}_g) \\ (\text{'BEEF'}, \text{CM}_y) \end{cases}$$

CM_b, the consensus message for BABE, is of the following format:

$$\text{CM}_b = \begin{cases} 1 & (\text{Auth}_C, r) \\ 2 & A_i \\ 3 & D \end{cases}$$

where

- *1* implies **next epoch data**: The Runtime issues this message on every first block of an epoch. The supplied authority set (Definition 63), Auth_C, and randomness (Definition 79), r, are used in the next epoch $\mathcal{E}_n + 1$.
- *2* implies **on disabled**: A 32-bit integer, A_i, indicating the individual authority in the current authority list that should be immediately disabled until the next authority set changes. This message initial intension was to cause an imediatly suspension of all authority functionality with the specified authority.
- *3* implies **next epoch descriptor**: These messages are only issued on configuration change and in the first block of an epoch. The supplied configuration data are intended to be used from the next epoch onwards.

  D is a varying datatype of the following format:

  $$D = \{(1, (c, 2_{\text{nd}}))$$

  where c is the probability that a slot will not be empty (Definition 69). It is encoded as a tuple of two unsigned 64-bit integers $(c_{\text{nominator}}, c_{\text{denominator}})$ which are used to compute the rational $c = \frac{c_{\text{nominator}}}{c_{\text{denominator}}}$.

  2_{nd} describes what secondary slot (Section 5.2.2.2), if any, is to be used. It is encoded as one-byte varying datatype:

  $$s_{2\text{nd}} = \begin{cases} 0 & -\&> ; & \text{no secondary slot} \\ 1 & -\&> ; & \text{plain secondary slot} \\ 2 & -\&> ; & \text{secondary slot with VRF output} \end{cases}$$

CM_g, the consensus message for GRANDPA, is of the following format:

$$\text{CM}_g = \begin{cases} 1 & (\text{Auth}_C, N_{\text{delay}}) \\ 2 & (m, \text{Auth}_C, N_{\text{delay}}) \\ 3 & \text{A}_i \\ 4 & N_{\text{delay}} \\ 5 & N_{\text{delay}} \end{cases}$$

where:

- N_{delay} is an unsigned 32-bit integer indicating how deep in the chain the announcing block must be before the change is applied.
- *1* implies **scheduled change**: Schedule an authority set change after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' is the block where the change is applied. The earliest digest of this type in a single block will be respected, unless a force change is present, in which case the force change takes precedence.
- *2* implies **forced change**: Schedule a forced authority set change after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, m + B')|$ where B' is the block where the change is applied. The earliest digest of this type in a block will be respected.

  Forced changes are explained further in Section 5.3.5.
- *3* implies **on disabled**: An index to the individual authority in the current authority list (Definition 63) that should be immediately disabled until the next authority set changes. When an authority gets disabled, the node should stop performing any authority functionality from that authority, including authoring blocks and casting GRANDPA votes for finalization. Similarly, other nodes should ignore all messages from the indicated authority which pertain to their authority role.
- *4* implies **pause**: A signal to pause the current authority set after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' is a block where the change is applied. Once applied, the authorities should stop voting.
- *5* implies **resume**: A signal to resume the current authority set after the given delay of $N_{\text{delay}} := |\text{SubChain}(B, B')|$ where B' is the block where the change is applied. Once applied, the authorities should resume voting.

The BEEFY protocol is still under construction. The following part will be updated in the future and certain information will be clarified.

CM_y, the consensus message for BEEFY (Section 5.5), is of the following format:

$$\text{CM}_y = \begin{cases} 1 & (V_B, V_i) \\ 2 & A_i \\ 3 & R \end{cases}$$

where:

- *1* implies that the remote **authorities have changed**. V_B is the array of the new BEEFY authorities's public keys and V_i is the identifier of the remote validator set.
- *2* implies **on disabled**: an index to the individual authorty in V_B that should be immediately disabled until the next authority change.
- *3* implies **MMR root**: a 32-byte array containing the MMR root.

5.2. Block Production

The Polkadot Host uses BABE protocol for block production. It is designed based on Ouroboros praos . 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 6 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 that do not include the most recent finalized blocks (Definition 6).

5.2.1. Preliminaries

5.2.1.1. Block Producer

A **block producer**, noted by $\mathcal{P}_j$, is a node running the Polkadot Host which is authorized to keep a transaction queue and which it gets a turn in producing blocks.

5.2.1.2. Block Authoring Session Key Pair

Block authoring session key pair (sk_j^s, pk_j^s) is an SR25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (Definition 152) and is used to sign the produced block as well as to compute its lottery values in Algorithm 6.

Definition 65. Epoch and Slot

A block production **epoch**, formally referred to as $\mathcal{E}$, is a period with a 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 a **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.

Substrate refers to an epoch as "session" in some places, however, epoch should be the preferred and official name for these periods.

Definition 66. Epoch and Slot Duration

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_configuration` (Section 26.8.1) at genesis. For a given block B, we use the notation s_B to refer to the slot during which B has been produced. Conversely, for slot s, $\mathcal{B}_c$ is the set of Blocks generated at slot s.

Definition 67 provides an iterator over the blocks produced during a specific epoch.

Definition 67. Epoch Subchain

By $\text{SubChain}(\mathcal{E}_n)$ for epoch $\mathcal{E}_n$, we refer to the path graph of BT containing 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 $\text{Longest-Chain}(BT)$.

Definition 68. Equivocation

A block producer **equivocates** if they produce more than one block at the same slot. The proof of equivocation are the given distinct headers that were signed by the validator and which include the slot number.

The Polkadot Host must detect equivocations committed by other validators and submit those to the Runtime as described in Section 26.8.6.

5.2.2. Block Production Lottery

The babe constant (Definition 69) is initialized at genesis to the value returned by calling `BabeApi_configuration` (Section 26.8.1). For efficiency reasons, it is generally updated by the Runtime through the *next config data* consensus message (Definition 64) in the digest of the first block of an epoch for the next epoch.

A block producer aiming to produce a block during $\mathcal{E}_n$ should run <algo-block-production-lottery>> 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 the epoch for whose slots the block producer is running the lottery.

In order to ensure consistent block production, BABE uses secondary slots in case no authority won the (primary) block production lottery. Unlike the lottery, secondary slot assignees are know upfront publically (Section 5.2.2.2). The Runtime provides information on how or if secondary slots are executed (Section 26.8.1), explained further in Section 5.2.2.2.

Definition 69. BABE Constant

The **BABE constant** is the probability that a slot will not be empty and used in the winning threshold calculation (Definition 70). It's expressed as a rational, (x, y), where x is the numerator and y is the denominator.

Definition 70. Winning Threshold

The **Winning threshold** denoted by $T_{\mathcal{E}_n}$ is the threshold that is used alongside the result of Algorithm 6 to decide if a block producer is the winner of a specific slot. $T_{\mathcal{E}_n}$ is calculated as follows:

$$A_w = \sum_{n=1} \text{Auth}_C(B)|)(w_A \in \text{Auth}_C(B)_n) T_{\mathcal{E}_n} : = 1 - (1-c)^{\frac{w_a}{A_w}}$$

where A_w is the total sum of all authority weights in the authority set (Definition 63) for epoch $\mathcal{E}_n$, w_a is the weight of the block author and $c \in (0, 1)$ is the BABE constant (Definition 69).

5.2.2.1. Primary Block Production Lottery

A block producer aiming to produce a block during $\mathcal{E}_n$ should run the Block-Production-Lottery algorithm to identify the slots it is awarded. These are the slots during which the block producer is

allowed to build a block. The session secret key, sk, is the block producer lottery secret key and n is the index of the epoch for whose slots the block producer is running the lottery.

Algorithm 6 Block-Production-Lottery

Require: sk

```
r ← Epoch-Randomness(n)
for i := 1 to sc_n do
    (π, d) ← VRF (r, i, sk)
    A[i] ← (d, π)
end for
return A
```

Algorithm 6. Block-Production-Lottery

where Epoch-Randomness is defined in (Definition 79), sc_n is defined in Definition 66 , VRF creates the BABE VRF transcript (Section 5.2.2.3) and e_i is the epoch index, retrieved from the Runtime (Section 26.8.1). s_k and p_k is the secret key respectively the public key of the authority. For any slot s in epoch n where $o < T_{\varepsilon_n}$ (Definition 70), the block producer is required to produce a block.

the secondary slots (Section 5.2.2.2) are running along side the primary block production lottery and mainly serve as a fallback to in case no authority was selected in the primary lottery.

5.2.2.2. Secondary Slots

Secondary slots work along side primary slot to ensure consistent block production, as described in Section 5.2.2. The secondary assignee of a block is determined by calculating a specific value, i_d, which indicates the index in the authority set (Definition 63). The corresponding authority in that set has the right to author a secondary block. This calculation is done for every slot in the epoch, $s \in sc_n$ (Definition 66).

$$p \leftarrow h(\mathrm{Enc}_{\mathrm{SC}}(r,\, s))i_d \leftarrow p \bmod A_l$$

where:

- r is the Epoch randomness (Definition 79).
- s is the slot number (Definition 65).
- $\mathrm{Enc}_{\mathrm{SC}}(\text{ellipsis})$ encodes its inner value to the corresponding SCALE value.
- $h(\text{ellipsis})$ creates a 256-bit Blake2 hash from its inner value.
- A_l is the lengths of the authority list (Definition 63).

If i_d points to the authority, that authority must claim the secondary slot by creating a BABE VRF transcript (Section 5.2.2.3). The resulting values o and p are then used in the Pre-Digest item (Definition 77). In case of secondary slots with plain outputs, respectively the Pre-Digest being of value *2*, the transcript respectively the VRF is skipped.

5.2.2.3. BABE Slot VRF transcript

The BABE block production lottery requires a specific transcript structure (Definition 150). That structure is used by both primary slots (Algorithm 6) and secondary slots (Section 5.2.2.2).

$t_1 \leftarrow \text{Transcript}(\text{'BABE'}) t_2 \leftarrow \text{append}(t_1, \text{'slot number'}, s) t_3 \leftarrow \text{append}(t_2, \text{'current epoch'}, e_i) t_4 \leftarrow \text{append}(t_3, \text{'chain randomness'}, r) t_5 \leftarrow \text{append}(t_4, \text{'vrf-nm-pk'}, p_k) t_6 \leftarrow \text{meta-ad}(t_5, \text{'VRFHash'}, \text{False}) t_7 \leftarrow \text{meta-ad}(t_6, 64_{le}, \text{True}) h \leftarrow \text{prf}(t_7, \text{False}) o = s_k \cdot hp \leftarrow \text{dleq\_prove}(t_7, h)$

The operators are defined in Definition 151, dleq_prove in Definition 148. The computed outputs, o and p, are included in the block Pre-Digest (Definition 77).

5.2.3. Slot Number Calculation

It is imperative for the security of the network that each block producer correctly determines the current slot numbers at a given time by regularly estimating the local clock offset in relation to the network (Definition 72).

The calculation described in this section is still to be implemented and deployed: For now, each block producer is required to synchronize its local clock using NTP instead. The current slot s is then calculated by $s = \frac{t_{\text{unix}}}{\mathcal{T}}$ where $\mathcal{T}$ is defined in Definition 65 and t_{unix} is defined in Definition 4. That also entails that slot numbers are currently not reset at the beginning of each epoch.

Polkadot does this synchronization without relying on any external clock source (e.g. through the or the). To stay in synchronization, each producer is therefore required to periodically estimate its local clock offset in relation to the rest of the network.

This estimation depends on the two fixed parameters k (Definition 73) and s_{cq} (Definition 74). These are chosen based on the results of a formal security analysis, currently assuming a $1s$ clock drift per day and targeting a probability lower than 0.5% for an adversary to break BABE in 3 years with resistance against a network delay up to $\frac{1}{3}$ of the slot time and a Babe constant (Definition 69) of $c = 0.38$.

All validators are then required to run Algorithm 8 at the beginning of each sync period (Definition 76) to update their synchronization using all block arrival times of the previous period. The algorithm should only be run once all the blocks in this period have been finalized, even if only probabilistically (Definition 73). The target slot to which to synchronize should be the first slot in the new sync period.

Definition 71. Slot Offset

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 the time continuum. As such, we have *Slot-Offset*$(s_i, s_i) = 0$.

It is imperative for the security of the network that each block producer correctly determines the

current slot numbers at a given time by regularly estimating the local clock offset in relation to the network (Definition 72).

Definition 72. Relative Time Synchronization

The **relative time synchronization** is a tuple of a slot number and a local clock timestamp $(s_{\text{sync}}, t_{\text{sync}})$ describing the last point at which the slot numbers have been synchronized with the local clock.

Algorithm 7 Slot-Time

Require: s

1: **return** $t_{\text{sync}} +$ Slot-Offset$(s_{\text{sync}}, s) \times T$

Algorithm 7. Slot-Time

where s is the slot number.

Algorithm 8 Median-Algorithm

Require: E, s_{sync}

1: $T_s \leftarrow \{\}$
2: **for** B **in** E_j **do**
3: $t^B_{\text{est}} \leftarrow T_B +$ Slot-Offset$(s_B, s_{\text{sync}}) \times T$
4: $T_s \leftarrow T_s \cup t^B_{\text{est}}$
5: **end for**
6: **return** Median(T_s)

Algorithm 8. Median-Algorithm

where

- $\mathfrak{C}$ is the sync period used for the estimate.
- s_{sync} is the slot time to estimate.
- Slot-Offset is defined in Algorithm 7.
- $\mathcal{T}$ is the slot duration defined in Definition 65.

Definition 73. Pruned Best Chain

The **pruned best chain** C^{rk} is the longest selected chain (Definition 8) with the last k Blocks pruned. We chose $k = 140$. The **last (probabilistic) finalized block** describes the last block in this pruned best chain.

Definition 74. *Chain Quality*

The **chain quality** s_{cq} represents the number of slots that are used to estimate the local clock offset. Currently, it is set to $s_{cq} = 3000$.

The prerequisite for such a calculation is that each producer stores the arrival time of each block (Definition 75) measured by a clock that is otherwise not adjusted by any external protocol.

Definition 75. *Block Arrival Time*

The **block arrival time** of block B for node j formally represented by T_B^j is the local time of node j when node j has received 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.

Definition 76. *Sync Period*

A is an interval at which each validator (re-)evaluates its local clock offsets. The first sync period $\mathfrak{E}_1$ starts just after the genesis block is released. Consequently, each sync period $\mathfrak{E}_i$ starts after $\mathfrak{E}_{i-1}$. The length of the sync period (Definition 74) is equal to $s_{qc}$and expressed in the number of slots.

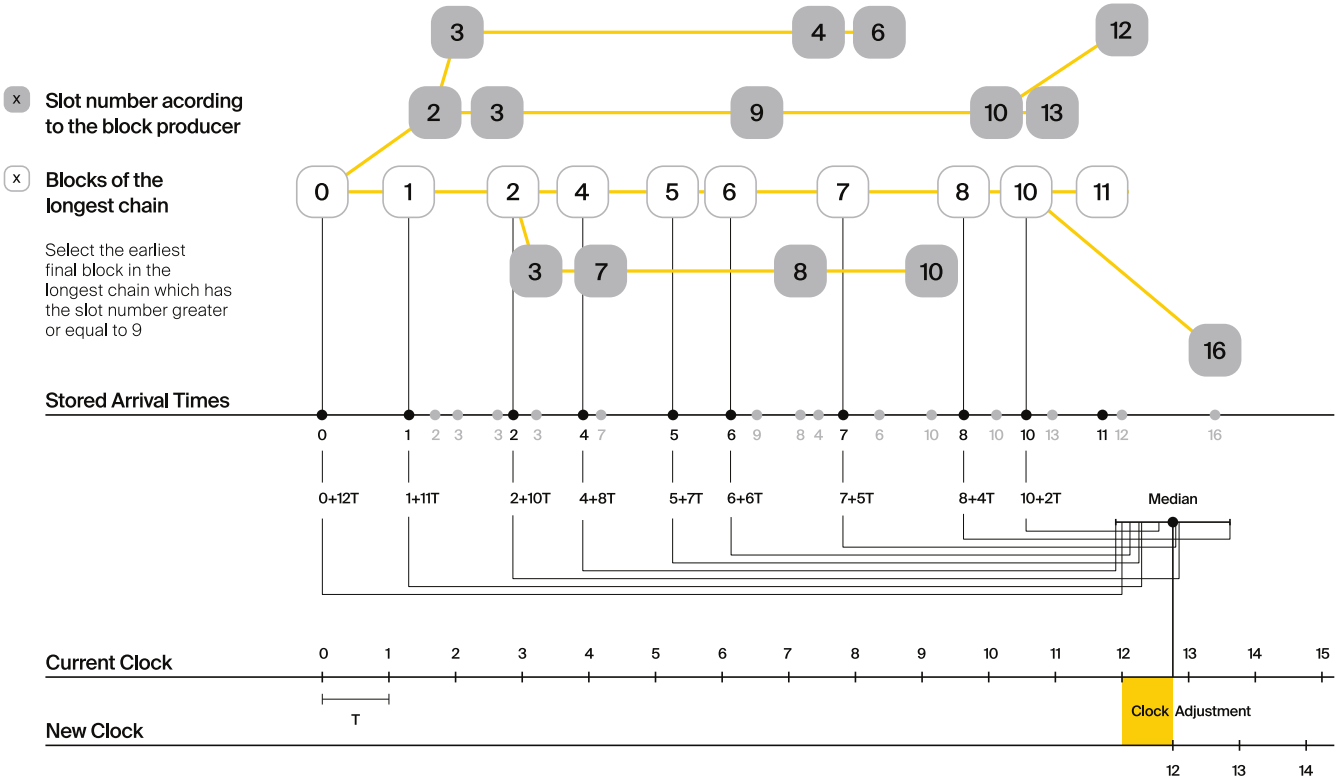


Figure 1. An exemplary result of Median Algorithm in first sync epoch with $s_{cq} = 9$ *and* $k = 1$.

5.2.4. Block Production

Throughout each epoch, each block producer should run Algorithm 9 to produce blocks during the slots it has been awarded during that epoch. The produced block needs to carry the *Pre-Digest* (Definition 77) as well as the *block signature* (Definition 78) as Pre-Runtime and Seal digest items.

Definition 77. Pre-Digest

The **Pre-Digest**, or BABE header, P, is a varying datatype of the following format:

$$P = \begin{cases} 1 \quad -\&> ; \quad (a_{\text{id}}, s, o, p) \\ 2 \quad -\&> ; \quad (a_{\text{id}}, s) \\ 3 \quad -\&> ; \quad (a_{\text{id}}, s, o, p) \end{cases}$$

where:

- *1* indicates a primary slot with VRF outputs, *2* a secondary slot with plain outputs and *3* a secondary slot with VRF outputs (Section 5.2.2). Plain outputs are no longer actively used and only exist for backwards compatibility reasons, respectively to sync old blocks.
- a_{id} is the unsigned 32-bit integer indicating the index of the authority in the authority set (Section 5.1.1) who authored the block.
- s is the slot number (Definition 65).
- o is VRF output (Algorithm 6 respectively Section 5.2.2.2).
- p is VRF proof (Algorithm 6 respectively Section 5.2.2.2).

The Pre-Digest must be included as a digest item of Pre-Runtime type in the header digest (Definition 29) $H_d(B)$.

Algorithm 9 Invoke-Block-Authoring

Require: sk, pk, n, BT
A ← Block-production-lottery(sk, n)
for s ← 1 **to** sc_n **do**
  Wait-Until(Slot-Time(s))
  (d, π) ← A[s]
  if d < τ **then**
    C_{Best} ← Longest-Chain(BT)
    B_s ← Build-Block(C_{Best})
    Add-Digest-Item(B_s, Pre-Runtime, E_{id}(BABE), $H_{BABE}(B_s)$)
    Add-Digest-Item(B_s, Seal, S_B)
    Broadcast-Block(B_s)
  end if
end for

Algorithm 9. Invoke-Block-Authoring

where BT is the current block tree, Block-Production-Lottery is defined in Algorithm 6 and Add-Digest-Item appends a digest item to the end of the header digest $H_d(B)$ (Definition 29).

Definition 78. Block Signature

The **Block Signature** S_B is a signature of the block header hash (Definition 30) and defined as

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

S_B should be included in $H_d(B)$ as the Seal digest item (Definition 29) of value:

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

in which, $E_{\text{id}}(\text{BABE})$ is the BABE consensus engine unique identifier (Definition 64). The Seal digest item is referred to as the **BABE Seal**.

5.2.5. Epoch Randomness

At the beginning of each epoch, $\mathcal{E}_n$ the host will receive the randomness seed $\mathcal{R}_{\mathcal{E}_{n+1}}$ (Definition 79) necessary to participate in the block production lottery in the next epoch $\mathcal{E}_{n+1}$ from the Runtime, through the consensus message (Definition 64) in the digest of the first block.

Definition 79. Randomness Seed

For epoch $\mathcal{E}$, there is a 32-byte $\mathcal{R}_\mathcal{E}$ computed based on the previous epochs VRF outputs. For $\mathcal{E}_0$ and $\mathcal{E}_1$, the randomness seed is provided in the genesis state (Section 26.8.1). For any further epochs, the randomness is retrieved from the consensus message (Definition 64).

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 10. Algorithm 11 runs as part of the verification process, when a node is importing a block.

Algorithm 10 Verify-Authorship-Right

Require: $\text{Head}_s(B)$

$s \leftarrow$ Slot-Number-At-Given-Time(T_B)
$E_c \leftarrow$ Current-Epoch()
$(D_1, \ldots, D_{|H_d(B)|}) \leftarrow H_d(B)$
$D_s \leftarrow D_{|H_d(B)|}$
$H_d(B) \leftarrow (D_1, \ldots, D_{|H_d(B)|-1})$ // remove the seal from the digest
$(\text{id}, \text{Sig}_B) \leftarrow \text{Dec}_{SC}(D_s)$
if id $\neq$ Seal-Id **then**
    error "Seal missing"
end if
AuthorID $\leftarrow$ AuthorityDirectory^{E_c}[$H_{BABE}(B)$.SingerIndex]
Verify-Signature(AuthorID, $H_h(B)$, Sig_B)
if $\exists B' \in BT : H_h(B) \neq H_h(B)$ **and** $s_B = s_B$ **and** SignerIndex$_x$ = SignerIndex$_x$ **then**
    error "Block producer is equivocating"
end if
Verify-Slot-Winner($(d_B, \pi_B), s_B$, AuthorID)

Algorithm 10. Verify-Authorship-Right

where

- $\text{Head}_s(B)$ is the header of the block that's being verified.
- T_B is B's arrival time (Definition 75).
- $H_d(B)$ is the digest sub-component (Definition 29) of $\text{Head}(B)$ (Definition 28).
- The Seal D_s is the last element in the digest array $H_d(B)$ as described in Definition 29.
- Seal-Id is the type index showing that a digest item (Definition 29) of varying type (Definition 163) is of type *Seal*.
- AuthorityDirectory$^{\mathcal{E}_c}$ is the set of Authority ID for block producers of epoch $\mathcal{E}_c$.
    1. AuthorId is the public session key of the block producer.
- BT is the pruned block tree (Definition 6).
- Verify-Slot-Winner is defined in Algorithm 11.

Algorithm 11 Verify-Slot-Winner

Require: B

$E_c \leftarrow$ Current-Epoch
$\rho \leftarrow$ Epoch-Randomness(ε)
Verify-VRF(ρ, $H_{BABE}(B).(d_B, \pi_B)$, $H_{BABE}(B).s$, c)
if $d_B \geq \tau$ **then**
    error "Block producer is not a winner of the slot"
end if

Algorithm 11. Verify-Slot-Winner

where

1. Epoch-Randomness is defined in Definition 79.
2. $H_{\text{BABE}}(B)$ is the BABE header defined in Definition 77.
3. (o, p) is the block lottery result for block B (Algorithm 6), respectively the VRF output (Section 5.2.2.3).
4. Verify-VRF is described in Section 10.4.
5. T_{ε_n} is the winning threshold as defined in Definition 70.

5.2.7. Block Building Process

The block building process is triggered by Algorithm 9 of the consensus engine which in turn runs Algorithm 12.

Algorithm 12 Build-Block

```
P_B ← Head(C_Best)
Head(B) ← (H_p ← H_h(P_B), H_i ← H_i(P_B) + 1, H_r ← ϕ, H_e ← ϕ, H_d ← ϕ)
Call-Runtime-Entry(Core_initialize_block, Head(B))
I-D ← Call-Runtime-Entry(BlockBuilder_inherent_extrinsics, Inherent-Data)
for E in I-D do
    Call-Runtime-Entry(BlockBuilder_apply_extrinsics, E)
end for
while not End-Of-Slot(s) do
    E ← Next-Ready-Extrinsic()
    R ← Call-Runtime-Entry(BlockBuilder_apply_extrinsics, E)
    if Block-Is-Full(R) then
        break
    end if
    if Should-Drop(R) then
        Drop(E)
    end if
    Head(B) ← Call-Runtime-Entry(BlockBuilder_finalize_block, B)
    B ← Add-Seal(B)
end while
```

Algorithm 12. Build-Block

where

- C_{Best} is the chain head at which the block should be constructed ("parent").
- s is the slot number.
- Head(B) is defined in Definition 28.
- Call-Runtime-Entry is defined in Definition 26.
- Inherent-Data is defined in Section 3.2.3.1.
- End-Of-Slot indicates the end of the BABE slot as defined Algorithm 8 respectively Definition 65.
- Next-Ready-Extrinsic indicates picking an extrinsic from the extrinsics queue (Definition 27).
- Block-Is-Full indicates that the maximum block size is being used.
- Should-Drop determines based on the result R whether the extrinsic should be dropped or remain in the extrinsics queue and scheduled for the next block. The *ApplyExtrinsicResult* (Definition 189) describes this behavior in more detail.
- Drop indicates removing the extrinsic from the extrinsic queue (Definition 27).
- Add-Seal adds the seal to the block (<<>>) before sending it to peers. The seal is removed again before submitting it to the Runtime.

5.3. Finality

The Polkadot Host uses GRANDPA Finality protocol to finalize blocks. Finality is obtained by consecutive rounds of voting by the 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 performs to successfully participate in the block-finalization process.

5.3.1. Preliminaries

Definition 80. GRANDPA Voter

A **GRANDPA Voter**, v, represented by a key pair $(K_v^{\text{pr}}, v_{\text{id}})$ where k_v^{pr} represents an *ed25519* private key, is a node running a GRANDPA protocol and broadcasting votes to finalize blocks in a Polkadot Host-based chain. The **set of all GRANDPA voters** for a given block B is indicated by $\mathbb{V}_B$. In that regard, we have [To do: change function name, only call at genesis, adjust V_B over the sections]

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

where `grandpa_authorities` is a function entry of the Runtime described in Section 26.7.1. We refer to $\mathbb{V}_B$ as $\mathbb{V}$ when there is no chance of ambiguity.

Analogously we say that a Polkadot node is a **non-voter node** for block B, if it does not own any of the key pairs in $\mathbb{V}_B$.

Definition 81. Authority Set Id

The **authority set Id** (id_V) is an incremental counter which tracks the amount of authority list changes that occurred (Definition 64). Starting with the value of zero at genesis, the Polkadot Host increments this value by one every time a **Scheduled Change** or a **Forced Change** occurs. The authority set Id is an unsigned 64-bit integer.

Definition 82. GRANDPA State

The **GRANDPA state**, GS, is defined as:

$$\text{GS} := \{\mathbb{V}, \text{id}_V, r\}$$

where:

- $\mathbb{V}$: is the set of voters.
- id_V: is the authority set ID (Definition 81).
- r: is the voting round number.

Definition 83. GRANDPA Vote

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 (Definition 30) and the block number (Definition 28).

Definition 84. Voting Rounds

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

Voting is done by means of broadcasting voting messages (Section 4.8.6) to the network. Validators inform their peers about the block finalized in round r by broadcasting a commit message (Algorithm 14).

Definition 85. Vote Signature

$\mathrm{Sign}_{v_i}^{r,\,\mathrm{stage}}$ refers to the signature of a voter for a specific message in a round and is formally defined as:

$$\mathrm{Sign}_{v_i}^{r,\,\mathrm{stage}} := \mathrm{Sig}_{\mathrm{ed25519}}(\mathrm{msg}, r, \mathrm{id}_{\mathbb{V}})$$

where:

- msg: is an byte array containing the message to be signed (Definition 83).
- r: is an unsigned 64-bit integer is the round number.
- $\mathrm{id}_{\mathbb{V}}$: is an unsigned 64-bit integer indicating the authority set Id (Definition 63).

Definition 86. Justification

The **justification** for block B in round r, $J^{r,\,\mathrm{stage}}(B)$, is a vector of pairs of the type:

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

in which either

$$B' \,\&>;\, = B$$

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

In all cases, $\mathrm{Sign}_{v_i}^{r,\,\mathrm{stage}}(B')$ is the signature (Definition 85) of voter $v_{\mathrm{id}} \in \mathbb{V}_B$ broadcasted during either the pre-vote (stage = pv) or the pre-commit (stage = pc) sub-round of round r. A **valid justification** must only contain up-to-one valid vote from each voter and must not contain more than two equivocatory votes from each voter.

Definition 87. Finalizing Justification

We say $J^{r,\,\mathrm{pc}}(B)$ **justifies the finalization** of $B' \geq B$ **for a non-voter node** n if the number of valid signatures in $J^{r,\,\mathrm{pc}}(B)$ for B' is greater than $\frac{2}{3}|\mathbb{V}_B|$.

Note that $J^{r,\,\mathrm{pc}}(B)$ can only be used by a non-voter node to finalize a block. In contrast, a voter node can only be assured of the finality (Definition 95) of block B by actively participating in the voting process. That is by invoking Algorithm 14.

The GRANDPA protocol dictates how an honest voter should vote in each sub-round, which is described by Algorithm 14. After defining what constitutes a vote in GRANDPA, we define how GRANDPA counts votes.

Definition 88. Equivocation

Voter v **equivocates** if they broadcast two or more valid votes to blocks 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 sub-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:

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

The Polkadot Host must detect equivocations committed by other validators and submit those to the Runtime as described in Section 26.7.2.

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 no match the current $\mathbb{V}$.
- $V_v^{r,\,\mathrm{stage}}$ is an equivocatory vote.

Definition 89. Set of Observed Direct Votes

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 90. *Set of Total Observed Votes*

We refer to **the set of total votes observed by voter v in sub-round *stage* of round r** by $V^{r,\,\text{stage}}_{\text{obs}(v)}$.

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

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

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 equivocator voters:

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

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

Definition 91. *Set of Total Potential Votes*

Let $V^{r,\,\text{stage}}_{\text{unobs}(v)}$ be the set of voters whose vote in the given stage has not been received. We define the **total number of potential votes for Block B in round r** to be:

$$\#\, V^{r,\,\text{stage}}_{\text{obs}(v),\,\text{pot}}(B) := \left|V^{r,\,\text{stage}}_{\text{obs}(v)}(B)\right| + \left|V^{r,\,\text{stage}}_{\text{unobs}(v)}\right| + \text{Min}\left(\frac{1}{3}|\mathbb{V}|, \left|\mathbb{V}\right| - |V^{r,\,\text{stage}}_{\text{obs}(v)}(B)| - |V^{r,\,\text{stage}}_{\text{unobs}(v)}|\right)$$

Definition 92. *Current Pre-Voted Block*

The current **pre-voted** block $B^{r,\,\text{pv}}_v$ also know as GRANDPA GHOST is the block chosen by Algorithm 17:

$$B^{r,\,\text{pv}}_v := \text{GRANDPA-GHOST}(r)$$

Finally, we define when a voter v sees 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 finalized in this round.

Definition 93. *Completable Round*

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

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

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

5.3.2. Initiating the GRANDPA State

In order to participate coherently in the voting process, a validator must initiate its state and sync it

with other active validators. In particular, considering that voting is happening in different distinct rounds where each round of voting is assigned a unique sequential round number r_v, it needs to determine and set its round counter r equal to the voting round r_n currently undergoing in the network. The mandated initialization procedure for the GRANDPA protocol for a joining validator is described in detail in Algorithm 13.

The process of joining a new voter set is different from the one of rejoining the current voter set after a network disconnect. The details of this distinction are described further in this section.

5.3.2.1. Voter Set Changes

A GRANDPA voter node which is initiating GRANDPA protocol as part of joining a new authority set is required to execute Algorithm 13. The algorithm mandates the initialization procedure for GRANDPA protocol.

The GRANDPA round number reset to 0 for every authority set change.

Voter set changes are signaled by Runtime via a consensus engine message (Section 5.1.2). When Authorities process such messages they must not vote on any block with a higher number than the block at which the change is supposed to happen. The new authority set should reinitiate GRANDPA protocol by executing Algorithm 13.

Algorithm 13 Initiate-Grandpa

Input: r_{last}, B_{last}

1: Last-Finalized-Block $\leftarrow B_{last}$
2: Best-Final-Candidate(0) $\leftarrow B_{last}$
3: GRANDPA-GHOST (0) $\leftarrow B_{last}$
4: Last-Completed-Round $\leftarrow 0$
5: $r_n \leftarrow 1$
6: Play-Grandpa-round(r_n)

Algorithm 13. Initiate-Grandpa

where B_{last} is the last block which has been finalized on the chain (Definition 95). r_{last} is equal to the latest round the voter has observed that other voters are voting on. The voter obtains this information through various gossiped messages including those mentioned in Definition 95. r_{last} is set to *0* if the GRANDPA node is initiating the GRANDPA voting process as a part of a new authority set. This is because the GRANDPA round number resets to *0* for every authority set change.

5.3.3. Rejoining the Same Voter Set

When a voter node rejoins the network after a disconnect from the voter set and with the condition that there has been no change to the voter set at the time of the disconnect, the node must continue performing the GRANDPA protocol at the same state as before getting disconnected from the network, ignoring any possible progress in GRANDPA finalization. Following reconnection, the node eventually gets updated to the current GRANDPA round and synchronizes its state with the rest of the voting set through the process called Catchup (Section 5.4.1).

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 14.

Algorithm 14 Play-Grandpa-Round

Require: (r)

$t_{r,v} \leftarrow$ Current local time
primary $\leftarrow$ Derive-Primary(r)
if $v =$ primary **then**
  Broadcast$(M_v^{r-1,\text{Fin}}$(Best-Final-Candidate$(r-1))$
  if Best-Final-Candidate$(r-1) \neq$ Last-Finalized-Block **then**
    Broadcast$(M_v^{r-1,\text{Prim}}$(Best-Final-Candidate$(r-1)))$
  end if
end if
Receive-Messages **until** Time $\geq t_{r,v} + 2 \times T$ **or** r **is** completable
$L \leftarrow$ Best-Final-Candidate$(r-1)$
$N \leftarrow$ Best-PreVote-Candidate(r)
Broadcast$(M_v^{r,\text{pv}}(N))$
Receive-Messages **until** $B_v^{r,\text{pv}} \geq L$ **and** (Time $\geq t_{r,v} + 4 \times T$ **or** r **is** completable)
Broadcast$(M_v^{r,\text{pc}}(B_v^{r,\text{pv}}))$
repeat
  Receive-Messages
  Attempt-To-Finalize-At-Round(r)
until r **is** completable **and** Finalizable(r) **and** Last-Finalized-Block $\geq$ Best-Final-Candidate$(r-1)$
Play-Grandpa-round$(r+1)$
repeat
  Receive-Messages
  Attempt-To-Finalize-At-Round(r)
until Last-Finalized-Block $\geq$ Best-Final-Candidate(r)
if Last-Completed-Round $< r$ **then**
  Last-Completed-Round $\leftarrow r$
end if

Algorithm 14. Play-Grandpa-Round

where

- T is sampled from a log-normal distribution whose mean and standard deviation are equal to the average network delay for a message to be sent and received from one validator to another.
- Derive-Primary is described in Algorithm 15.
- The condition of *completablitiy* is defined in Definition 93.
- Best-Final-Candidate function is explained in Algorithm 16.
- Attempt-To-Finalize-At-Round(r) is described in Algorithm 19.
- Finalizable is defined in Algorithm 20.

Algorithm 15 Derive-Primary

Input: r

return $r \bmod |V|$

Algorithm 15. Derive-Primary

where r is the GRANDPA round whose primary is to be determined.

Algorithm 16 Best-Final-Candidate

Input: r

$B_v^{r,pv} \leftarrow$ GRANDPA-GHOST (r)
if $r = 0$ **then**
  return $B_v^{r,pv}$
else
  $C \leftarrow \{B' \in B' \geq B_v^{r,pv} \mid \#V_{obv(v),pot}^{r,pc}(B') > \frac{2}{3}|V|\}$
  if $C = \phi$ **then**
    return $B_v^{r,pv}$
  else
    return $E \in C : H_n(E) = \max(H_n(B') \mid B' \in C)$
  end if
end if

Algorithm 16. Best-Final-Candidate

where $\#V_{obv(v),\ pot}^{r,\ pc}$ is defined in Definition 91.

Algorithm 17 GRANDPA-GHOST

Input: r

if $r = 0$ **then**
  $G \leftarrow B_{last}$
else
  $L \leftarrow$ Best-Final-Candidate$(r-1)$
  $G = \{B > L \mid \#V_{obs(v)}^{r,pv}(B) \geq \frac{2}{3}|V|\}$
  if $G = \phi$ **then**
    $G \leftarrow L$
  else
    $G \in G \mid H_n(G) = \max(H_n(B) \mid B \in G)$
  end if
end if
return G

Algorithm 17. GRANDPA-GHOST

where

- B_{last} is the last block which has been finalized on the chain (Definition 95).
- $\#V_{\text{obs}(v)}^{r,\ pv}(B)$ is defined in Definition 90.

Algorithm 18 Best-PreVote-Candidate

Input: r

$B_v^{r,pv} \leftarrow$ GRANDPA-GHOST (r)
if Received$(M_{v_{\text{primary}}}^{r,\text{prim}}(B))$ **and** $B_v^{r,pv} \geq B > L$ **then**
  $N \leftarrow B$
else
  $N \leftarrow B_v^{r,pv}$
end if

Algorithm 18. Best-PreVote-Candidate

Algorithm 19 Attempt-To-Finalize-At-Round

Require: (r)

```
L ← Last-Finalized-Block
E ← Best-Final-Candidate(r)
if E ≠ L and V^{r,pc}_{obs(v)}(E) > 2/3|V| then
    Last-Finalized-Block ← E
    if M_v^{r,Fin}(E) ∉ Received-Messages then
        Broadcast(M_v^{r,Fin}(E))
        return
    end if
end if
```

Algorithm 19. Attempt-To-Finalize-At-Round

Algorithm 20 Finalizable

Require: (r)

```
if r is not Completable then
    return False
end if
G ← GRANDPA-GHOST(J^{r,pv}(B))
if G = ϕ then
    return False
end if
E_r ← Best-Final-Candidate(r)
if E_r ∋ ϕ and Best-Final-Candidate(r − 1) ≠ E_r ≠ G then
    return True
else
    return False
end if
```

Algorithm 20. Finalizable

where the condition for *completability* is defined in Definition 93.

Note that we might not always succeed in finalizing our best final candidate due to the possibility of equivocation. We might even not finalize anything in a round (although Algorithm 14 prevents us from moving to the round $r+1$ before finalizing the best final candidate of round $r-1$) The example in Definition 94 serves to demonstrate a situation where the best final candidate of a round cannot be finalized during its own round:

Definition 94. Unfinalized Candidate

Let us assume that we have 100 voters and there are two blocks in the chain ($B_1 < B_2$). At round 1, we get 67 pre-votes for B_2 and at least one pre-vote for B_1 which means that GRANDPA-GHOST$(1) = B_2$.

Subsequently, potentially honest voters who could claim not seeing all the pre-votes for B_2 but receiving the pre-votes for B_1 would pre-commit to B_1. In this way, we receive 66 pre-commits for B_1 and 1 pre-commit for B_2. Henceforth, we finalize B_1 since we have a threshold commit (67 votes) for B_1.

At this point, though, we have `Best-Final-Candidate`$(r) = B_2$ as $\#\, V^{r,\ \text{stage}}_{\text{obs}(v),\ \text{pot}}(B_2) = 67$ and $2 > 1$.

However, at this point, the round is already completable as we know that we have `GRANDPA-GHOST`$(1) = B_2$ as an upper limit on what we can finalize and nothing greater than B_2 can be finalized at $r = 1$. Therefore, the condition of Algorithm 14 is satisfied and we must proceed to round 2.

Nonetheless, we must continue to attempt to finalize round *1* in the background as the condition of Algorithm 19 has not been fulfilled.

This prevents us from proceeding to round 3 until either:

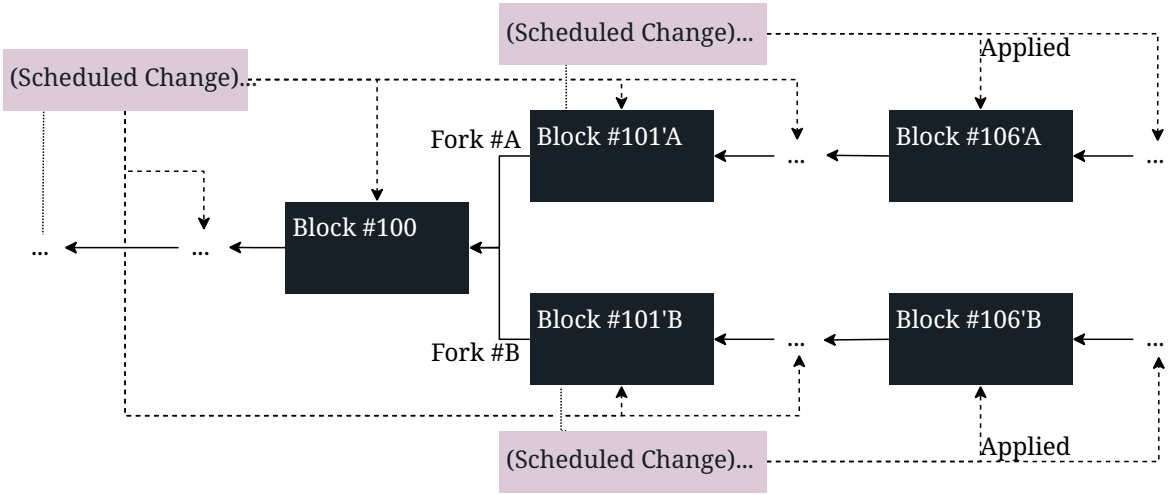
- We finalize B_2 in round 2, or
- We receive an extra pre-commit vote for B_1 in round 1. This will make it impossible to finalize B_2 in round 1, no matter to whom the remaining pre-commits are going to be cast for (even with considering the possibility of 1/3 of voter equivocating) and therefore we have `Best-Final-Candidate`$(r) = B_1$.

Both scenarios unblock Algorithm 14, `Last-Finalized-Block` $\geq$ `Best-Final-Candidate`$(r-1)$ albeit in different ways: the former with increasing the `Last-Finalized-Block` and the latter with decreasing `Best-Final-Candidate`$(r-1)$.

5.3.5. Forced Authority Set Changes

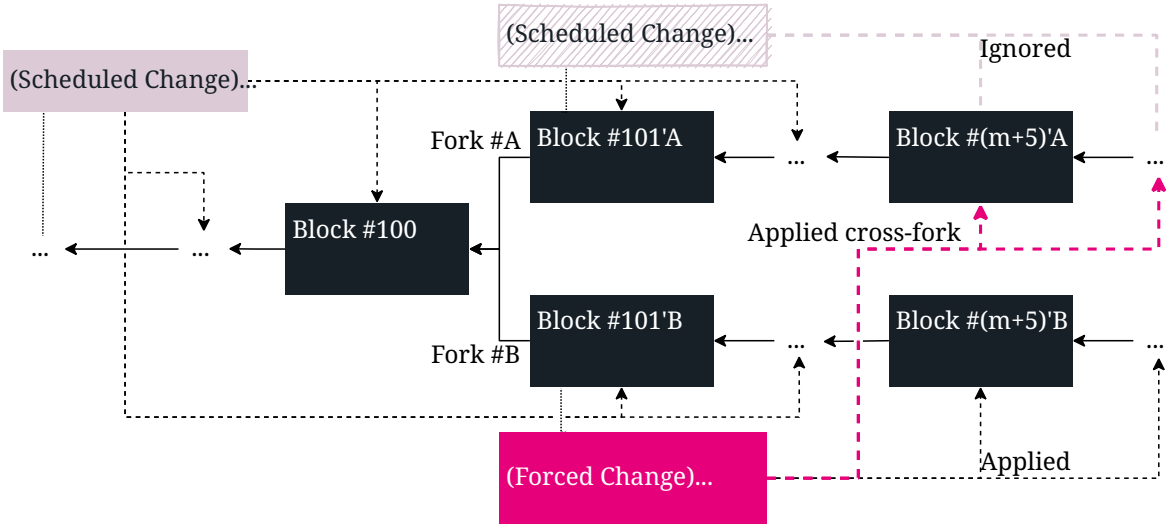
In a case of emergency where the Polkadot network is unable to finalize blocks, such as in an event of mass validator outage, the Polkadot governance mechanism must enact a forced change, which the Host must handle in a specific manner. Given that in such a case finality cannot be relied on, the Host must detect the forced change (Definition 64) in a (valid) block and apply it to all forks.

The $m \in CM_g$, which is specified by the governance mechanism, defines the starting block at which N_{delay} is applied. This provides some degree of probabilistic consensus to the network with the assumption that the forced change was received by most participants and that finality can be continued.



Text is not SVG - cannot display

Figure 2. Applying a scheduled change



Text is not SVG - cannot display

Figure 3. Applying a forced change

5.4. Block Finalization

Definition 95. Finalized

A Polkadot relay chain node n should consider block B as **finalized** if any of the following criteria hold for $B' \geq B$:

- $V^{r,\,\mathrm{pc}}_{\mathrm{obs}(n)}(B') > \frac{2}{3}|\mathbb{V}_{B'}|$.
- It receives a $M^{r,\,\mathrm{Fin}}_{v}(B')$ message in which $J^r(B)$ justifies the finalization (Definition 86).
- It receives a block data message for B' with $\mathrm{Just}(B')$ (Section 3.3.1.1) which justifies the finalization.

for:

- Any round r if the node n is *not* a GRANDPA voter.
- Only for round r for which the node n has invoked Algorithm 14 and round $r+1$ if n is a GRANDPA voter and has already caught up to its peers according to the process described in Section Section 5.4.1.

Note that all Polkadot relay chain nodes are supposed to process GRANDPA commit messages regardless of their GRANDPA voter status.

5.4.1. Catching up

When a Polkadot node (re)joins the network during the process described in [chapter-bootstrapping], it requests the history of state transitions in the form of blocks, which it is missing.

Nonetheless, the process is different for a GRANDPA voter node. When a voter node joins the network, it needs to gather the justification (Definition 86) of the rounds it has missed. Through this process, they can safely join the voting process of the current round, on which the voting is taking place.

5.4.1.1. Sending the catch-up requests

When a Polkadot voter node has the same authority list as a peer voter node who is reporting a higher number for the *finalized round* field, it should send a catch-up request message (Definition 48) to the reporting peer. This will allow the node to to catch up to the more advanced finalized round, provided that the following criteria hold:

- The peer node is a GRANDPA voter, and:
- The last known finalized round for the Polkadot node is at least 2 rounds behind the finalized round for the peer.

5.4.1.2. Processing the catch-up requests

Only GRANDPA voter nodes are required to respond to the catch-up requests. Additionally, it is only GRANDPA voters who are supposed to send catch-up requests. As such GRANDPA voters could safely ignore the catch-up requests from non-voter nodes. When a GRANDPA voter node receives a catch-up request message, it needs to execute Algorithm 21. Note: a voter node should not respond to catch-up requests for rounds that are actively being voted on, those are the rounds for which

Algorithm 14 is not concluded.

Algorithm 21 Process-Catchup-Request

Input: $M_{i,v}^{\text{Cat-q}}(\text{id}_V, r)$

```
if M^Cat-q_{i,v}(id_V, r).id_V ≠ id_V then
    error "Catching up on different set"
end if
if i ∉ P then
    error "Requesting catching up from a non-peer"
end if
if r > Last-Completed-Round then
    error "Catching up on a round in the future"
end if
Send(i, M^Cat-s_{v,i}(id_V, r))
```

Algorithm 21. Process-Catchup-Request

where

- $M_{i,\,v}^{\text{Cat}-q}(\text{id}_V, r)$ is the catch-up message received from peer i (Definition 48).
- id_V is the voter set id with which the serving node is operating
- r is the round number for which the catch-up is requested for.
- $\mathbb{P}$ is the set of immediate peers of node v.
- `Last-Completed-Round` is initiated in Algorithm 13 and gets updated by Algorithm 14.
- $M_{v,\,i}^{\text{Cat}-s}(\text{id}_V, r)$ is the catch-up response (Definition 49).

5.4.1.3. Processing catch-up responses

A Catch-up response message contains critical information for the requester node to update their view on the active rounds which are being voted on by GRANDPA voters. As such, the requester node should verify the content of the catch-up response message and subsequently updates its view of the state of the finality of the Relay chain according to Algorithm 22.

Algorithm 22 Process-Catchup-Response

Input: $M_{v,i}^{\text{Cat-s}}(\text{id}_V, r)$

$M_{v,i}^{\text{Cat-s}}(\text{id}_V, r).\text{id}_V, r, J^{r,pv}(B), J^{r,pc}(B), H_h(B'), H_i(B') \leftarrow \text{Dec}_{SC}(M_{v,i}^{\text{Cat-s}}(\text{id}_V, r))$
if $M_{v,i}^{\text{Cat-s}}(\text{id}_V, r).\text{id}_V \neq \text{id}_V$ **then**
  error "Catching up on different set"
end if
if $r \neq$ Leading-Round **then**
  error "Catching up in to the past"
end if
if $J^{r,pv}(B)$ **is not** valid **then**
  error "Invalid pre-vote justification"
end if
if $J^{r,pc}(B)$ **is not** valid **then**
  error "Invalid pre-commit justification"
end if
$G \leftarrow$ GRANDPA-GHOST $(J^{r,pv}(B))$
if $G = \phi$ **then**
  error "GHOST-less Catch-up"
end if
if r **is not** completable **then**
  error "Catch-up round is not completable"
end if
if $J^{r,pc}(B)$ justifies B' finalization **then**
  error "Unjustified Catch-up target finalization"
end if
Last-Completed-Round $\leftarrow r$
if $i \in V$ **then**
  Play-Grandpa-round$(r+1)$
end if

Algorithm 22. Process-Catchup-Response

where $M_{v,i}^{\text{Cat-s}}(\text{id}_V, r)$ is the catch-up response received from node v (Definition 49).

5.5. Bridge design (BEEFY)

The BEEFY protocol is currently in early development and subject to change. The specification has not been completed yet.

The BEEFY (Bridge Effiency Enabling Finality Yielder) is a secondary protocol to GRANDPA to support efficient bridging between the Polkadot network (relay chain) and remote, segregated blockchains, such as Ethereum, which were not built with the Polkadot interchain operability in mind. The protocol allows participants of the remote network to verify finality proofs created by the Polkadot relay chain validators. In other words: clients in the Ethereum network should able to verify that the Polkadot network is at a specific state.

Storing all the information necessary to verify the state of the remote chain, such as the block headers, is too expensive. BEEFY stores the information in a space-efficient way and clients can request additional information over the protocol.

5.5.1. Preliminaries

Definition 96. Merkle Mountain Ranges

Merkle Mountain Ranges, **MMR**, are used as an efficient way to send block headers and signatures to light clients.

MMRs have not been defined yet.

Definition 97. Statement

The **statement** is the same piece of information which every relay chain validator is voting on. Namely, the MMR root of all the block header hashes leading up to the latest, finalized block.

Definition 98. Witness Data

Witness data contains the statement (Definition 97), an array indicating which validator of the Polkadot network voted for the statement (but not the signatures themselves) and a MMR root of the signatures. The indicators of which validator voted for the statement are just claims and provide no proofs. The network message is defined in Definition 53 and the relayer saves it on the chain of the remote network.

Definition 99. Light Client

A **light client** is an abstract entity in a remote network such as Ethereum. It can be a node or a smart contract with the intent of requesting finality proofs from the Polkadot network. A light client reads the witness data (Definition 98 from the chain, then requests the signatures directly from the relayer in order to verify those.

The light client is expected to know who the validators are and has access to their public keys.

Definition 100. Relayer

A **relayer** (or "prover") is an abstract entity which takes finality proofs from the Polkadot network and makes those available to the light clients. Inherently, the relayer tries to convince the light clients that the finality proofs have been voted for by the Polkadot relay chain validators. The relayer operates offchain and can for example be a node or a collection of nodes.

5.5.2. Voting on Statements

The Polkadot Host signs a statement (Definition 97) and gossips it as part of a vote (Definition 51) to its peers on every new, finalized block. The Polkadot Host uses ECDSA for signing the statement, since Ethereum has better compatibility for it compared to SR25519 or ED25519.

5.5.3. Committing Witnesses

The relayer (Definition 100) participates in the Polkadot network by collecting the gossiped votes (Definition 51). Those votes are converted into the witness data structure (Definition 98). The relayer saves the data on the chain of the remote network. The occurrence of saving witnesses on remote networks is undefined.

5.5.4. Requesting Signed Commitments

A light client (Definition 99) fetches the witness data (Definition 98) from the chain. Once the light client knows which validators apparently voted for the specified statement, it needs to request the signatures from the relayer to verify whether the claims are actually true. This is achieved by requesting signed commitments (Definition 52).

How those signed commitments are requested by the light client and delivered by the relayer varies among networks or implementations. On Ethereum, for example, the light client can request the signed commitments in form of a transaction, which results in a response in form of a transaction.

Chapter 6. Availability & Validity

Polkadot serves as a replicated shared-state machine designed to resolve scalability issues and interoperability among blockchains. The validators of Polkadot execute transactions and participate in the consensus of Polkadots primary chain, the so called relay chain. Parachains are independent networks that maintain their own state and are connected to the relay chain. Those parachains can take advantage of the relay chain consensus mechanism, including sending and receiving messages to and from other parachains. Parachain nodes that send parachain blocks, known as candidates, to the validators in order to be included in relay chain are referred to as collators.

The Polkadot relay chain validators are responsible for guaranteeing the validity of both relay chain and parachain blocks. Additionally, the validators are required to keep enough parachain blocks that should be included in the relay chain available in their local storage in order to make those retrievable by peers, who lack the information, to reliably confirm the issued validity statements about parachain blocks. The Availability & Validity (AnV) protocol consists of multiple steps for successfully upholding those responsibilities.

Parachain blocks themselves are produced by collators (Section 6.1), whereas the relay chain validators only verify their validity (and later, their availability). It is possible that the collators of a parachain produces multiple parachain block candidates for a child of a specific block. Subsequently, they send the block candidates to the the relay chain validators who are assigned to the specific parachain. The assignment is determined by the Runtime (Section 6.2). Those validators are then required to check the validity of submitted candidates (Section 6.3), then issue and collect statements (Section 6.2.1) about the validity of candidates to other validators. This process is known as candidate backing. Once a candidate meets a specified criteria for inclusion, the selected relay chain block author then choses any of the backed candidate for each parachain and includes those into the relay chain block (Section 6.2.2).

Every relay chain validator must fetch the proposed candidates and issue votes on whether they have the candidate saved in their local storage, so called availability votes (Section 6.4.1), then also collect the votes sent by other validators and include them in the relay chain state (Section 6.2.2). This process ensures that only relay chain blocks get finalized where each candidate is available on enough nodes of validators.

Parachain candidates contained in non-finalized relay chain blocks must then be retrieved by a secondary set of relay chain validators, unrelated from the candidate backing process, who are randomly assigned to determine the validity of specific parachains based on a VRF lottery and are then required to vote on the validity of those candidates. This process is known as approval voting (Section 6.5). If a validator does not have the candidate data, it must recover the candidate data (Section 6.4.2).

6.1. Collations

Collations are proposed candidates Section 6.8.2 to the Polkadot relay chain validators. The Polkodat network protocol is agnostic on what candidate productionis mechanism each parachain uses and does not specify or mandate any of such production methods (e.g. BABE-GRANDPA, Aura, etc). Furthermore, the relay chain validator host implementation itself does not directly interpret or

process the internal transactions of the candidate, but rather rely on the parachain Runtime to validate the candidate (Section 6.3). Collators, which are parachain nodes which produce candidate proposals and send them to the relay chain validator, must prepare pieces of data (Definition 101) in order to correctly comply with the requirements of the parachain protocol.

Definition 101. Collation

A collation is a datastructure which contains the proposed parachain candidate, including an optional validation parachain Runtime update and upward messages. The collation datastructure, C, is a datastructure of the following format:

$$C = (M, H, R, h, P, p, w) M = (u_n, \text{ Unknown character Unknown character Unknown character } u_m) H = (z_n, \text{ Unknown character Unknown character Unknown character } z_m)$$

where

- M is an array of upward messages (Section 6.8.8), u, interpreted by the relay chain itself.
- H is an array of outbound horizontal messages (Section 6.8.10), z, interpreted by other parachains.
- R is an *Option* type (Definition 164) which can contain a parachain Runtime update. The new Runtime code is an array of bytes.
- h is the head data (Section 6.8.4) produced as a result of execution of the parachain specific logic.
- P is the PoV block (Section 6.8.3).
- p is an unsigned 32-bit integer indicating the number of processed downward messages (Section 6.8.9).
- w is an unsigned 32-bit integer indicating the mark up to which all inbound HRMP messages have been processed by the parachain.

6.2. Candidate Backing

The Polkadot validator receives an arbitrary number of parachain candidates with associated proofs from untrusted collators. The assigned validators of each parachain (Section 6.8.7) must verify and select a specific quantity of the proposed candidates and issue those as backable candidates to its peers. A candidate is considered backable when at least 2/3 of all assigned validators have issued a *Valid* statement about that candidate, as described in Section 6.2.1. Validators can retrieve information about assignments via the Runtime APIs Section 26.6.2 respectively Section 26.6.3.

6.2.1. Statements

The assigned validator checks the validity of the proposed parachains blocks (Section 6.3) and issues *Valid* statements (Definition 102) to its peers if the verification succeeded. Broadcasting failed verification as *Valid* statements is a slashable offense. The validator must only issue one *Seconded* statement, based on an arbitrary metric, which implies an explicit vote for a candidate to be included in the relay chain.

This protocol attempts to produce as many backable candidates as possible, but does not attempt to determine a final candidate for inclusion. Once a parachain candidate has been seconded by at least one other validator and enough Valid statements have been issued about that candidate to meet the 2/3 quorum, the candidate is ready to be included in the relay chain (Section 6.2.2).

The validator issues validity statements votes in form of a validator protocol message (Definition 114).

Definition 102. Statement

A statement, S, is a datastructure of the following format:

$$S = (d, A_i, A_s) d = \begin{cases} 1 & -\&>; \ C_r \\ 2 & -\&>; \ C_h \end{cases}$$

where

- d is a varying datatype where *1* indicates that the validator "seconds" a candidate, meaning that the candidate should be included in the relay chain, followed by the committed candidate receipt (Definition 105), C_r. *2* indicates that the validator has deemed the candidate valid, followed by the candidate hash.
- C_h is the candidate hash.
- A_i is the validator index in the authority set that signed this statement.
- A_s is the signature of the validator.

6.2.2. Inclusion

The Polkadot validator includes the backed candidates as parachain inherent data (Definition 103) into a block as described Section 3.2.3. The relay chain block author decides on whatever metric which candidate should be selected for inclusion, as long as that candidate is valid and meets the validity quorum of 2/3+ as described in Section 6.2.1. The candidate approval process (Section 6.5) ensures that only relay chain blocks are finalized where each candidate for each availability core meets the requirement of 2/3+ availability votes.

Definition 103. *Parachain Inherent Data*

The parachain inherent data contains backed candidates and is included when authoring a relay chain block. The datastructure, I, is of the following format:

$$I = (A, T, D, P_h) T = (C_0, \text{Unknown character} \text{Unknown character} \text{Unknown character} C_n) D = (\cdot d_n, \text{Unknown character} \text{Unknown character} \text{Unknown character} d_{10}) C = (R, V, i) V = (a_0, \text{Unknown character} \text{Unknown character} \text{Unknown character} a_{10}) a = \begin{cases} 1 & \text{<} 8 \text{>}; \ s \\ 2 & \text{<} 8 \text{>}; \ s \end{cases} A = (L_0, \text{Unknown character} \text{Unknown character} \text{Unknown character} L_{10}) L = (b, v_i, s)$$

where

- A is an array of signed bitfields by validators claiming the candidate is available (or not). The array must be sorted by validator index corresponding to the authority set (Definition 63).
- T is an array of backed candidates for inclusing in the current block.
- D is an array of disputes.
- P_h is the parachain parent head data (Section 6.8.4).
- d is a dispute statement (Section 6.7.2.1).
- R is a committed candidate receipt (Definition 105).
- V is an array of validity votes themselves, expressed as signatures.
- i is a bitfield of indices of the validators within the validator group (Section 6.8.7).
- a is either an implicit or explicit attestation of the validity of a parachain candidate, where *1* implies an implicit vote (in correspondence of a *Seconded* statement) and *2* implies an explicit attestation (in correspondence of a *Valid* statement). Both variants are followed by the signature of the validator.
- s is the signature of the validator.
- b the availability bitfield (Section 6.4.1).
- v_i is the validator index of the authority set (Definition 63).

Definition 104. *Candidate Receipt*

A candidate receipt, R, contains information about the candidate and a proof of the results of its execution. It's a datastructure of the following format:

$$R = (D, C_h)$$

where D is the candidate descriptor (Definition 106) and C_h is the hash of candidate commitments (Definition 107).

Definition 105. Committed Candidate Receipt

The committed candidate receipt, R, contains information about the candidate and the the result of its execution that is included in the relay chain. This type is similar to the candidate receipt (Definition 104), but actually contains the execution results rather than just a hash of it. It's a datastructure of the following format:

$$R = (D, C)$$

where D is the candidate descriptor (Definition 106) and C is the candidate commitments (Definition 107).

Definition 106. Candidate Descriptor

The candidate descriptor, D, is a unique descriptor of a candidate receipt. It's a datastructure of the following format:

$$D = (p, H, C_i, V, B, r, s, p_h, R_h)$$

where

- p is the parachain Id (Section 6.8.5).
- H is the hash of the relay chain block the candidate is executed in the context of.
- C_i is the collators public key.
- V is the hash of the persisted validation data (Definition 199).
- B is the hash of the PoV block.
- r is the root of the block's erasure encoding Merkle tree.
- s the collator signature of the concatenated components p, H, R_h and B.
- p_h is the hash of the parachain head data (Section 6.8.4) of this candidate.
- R_h is the hash of the parachain Runtime.

Definition 107. Candidate Commitments

The candidate commitments, C, is the result of the execution and validation of a parachain (or parathread) candidate whose produced values must be committed to the relay chain. Those values are retrieved from the validation result (Definition 109). A candidate commitment is a datastructure of the following format:

$$C = (M_u, M_h, R, h, p, w)$$

where:

- M_u is an array of upward messages sent by the parachain. Each individual message, m, is an array of bytes.
- M_h is an array of individual outbound horizontal messages (Section 6.8.10) sent by the parachain.
- R is an *Option* value (Definition 164) that can contain a new parachain Runtime in case of an update.
- h is the parachain head data (Section 6.8.4).
- p is a unsigned 32-bit integer indicating the number of downward messages that were processed by the parachain. It is expected that the parachain processes the messages from first to last.
- w is a unsigned 32-bit integer indicating the watermark which specifies the relay chain block number up to which all inbound horizontal messages have been processed.

6.3. Candidate Validation

Received candidates submitted by collators and must have its validity verified by the assigned Polkadot validators. For each candidate to be valid, the validator must successfully verify the following conditions in the following order:

1. The candidate does not exceed any parameters in the persisted validation data (Definition 199).
2. The signature of the collator is valid.
3. Validate the candidate by executing the parachain Runtime (Section 6.3.1).

If all steps are valid, the Polkadot validator must create the necessary candidate commitments (Definition 107) and submit the appropriate statement for each candidate (Section 6.2.1).

6.3.1. Parachain Runtime

Parachain Runtimes are stored in the relay chain state, and can either be fetched by the parachain Id or the Runtime hash via the relay chain Runtime API as described in Section 26.6.7 and Section 26.6.8 respectively. The retrieved parachain Runtime might need to be decompressed based on the magic identifier as described in Section 6.3.2.

In order to validate a parachain block, the Polkadot validator must prepare the validation parameters (Definition 108), then use its local Wasm execution environment (Section 3.1.2) to

execute the validate_block parachain Runtime API by passing on the validation parameters as an argument. The parachain Runtime function returns the validation result (Definition 109).

Definition 108. Validation Parameters

The validation parameters structure, P, is required to validate a candidate against a parachain Runtime. It's a datastructure of the following format:

$$P = (h, b, B_i, S_r)$$

where

- h is the parachain head data (Section 6.8.4).
- b is the block body (Section 6.8.3).
- B_i is the latest relay chain block number.
- S_r is the relay chain block storage root (Section 2.1.4).

Definition 109. Validation Result

The validation result is returned by the `validate_block` parachain Runtime API after attempting to validate a parachain block. Those results are then used in candidate commitments (Definition 107), which then will be inserted into the relay chain via the parachain inherent data (Definition 103). The validation result, V, is a datastructure of the following format:

$$V = (h, R, M_u, M_h, p, w) M_u = (m_0, \text{ Unknown character Unknown character Unknown character } m_n) M_h = (t_0, \text{ Unknown character Unknown character Unknown character } t_n)$$

where

- h is the parachain head data (Section 6.8.4).
- R is an *Option* value (Definition 164) that can contain a new parachain Runtime in case of an update.
- M_u is an array of upward messages sent by the parachain. Each individual message, m, is an array of bytes.
- M_h is an array of individual outbound horizontal messages (Section 6.8.10) sent by the parachain.
- p is a unsigned 32-bit integer indicating the number of downward messages that were processed by the parachain. It is expected that the parachain processes the messages from first to last.
- w is a unsigned 32-bit integer indicating the watermark which specifies the relay chain block number up to which all inbound horizontal messages have been processed.

6.3.2. Runtime Compression

Runtime compression is not documented yet.

6.4. Availability

6.4.1. Availability Votes

The Polkadot validator must issue a bitfield (Section 6.8.12) which indicates votes for the availability of candidates. Issued bitfields can be used by the validator and other peers to determine which backed candidates meet the 2/3+ availability quorum.

Candidates are inserted into the relay chain in form of parachain inherent data (Section 6.2.2) by a block author. A validator can retrieve that data by calling the appropriate Runtime API entry (Section 26.6.3), then create a bitfield indicating for which candidate the validator has availability data stored and broadcast it to the network (Definition 118). When sending the bitfield distrubtion message, the validator must ensure B_h is set approriately, therefore clarifying to which state the bitfield is referring to, given that candidates can vary based on the chain fork.

Missing availability data of candidates must be recovered by the validator as described in Section 6.4.2. If previously issued bitfields are no longer accurate, i.e. the availability data has been recovered or the candidate of an availability core has changed, the validator must create a new bitfield and broadcast it to the network. Candidates must be kept available by validators for a specific amount of time. If a candidate does not receive any backing, validators should keep it available for about one hour, in case the state of backing does change. Backed and even approved candidates (Section 6.5) must be kept by validators for about 25 hours, since disputes (Section 6.6) can occur and the candidate needs to be checked again.

The validator issues availability votes in form of a validator protocol message (Definition 115).

6.4.2. Candidate Recovery

The availability distribution of the Polkadot validator must be able to recover parachain candidates that the validator is assigned to, in order to determine whether the candidate should be backed (Section 6.2) respectively whether the candidate should be approved (Section 6.5). Additionally, peers can send availability requests as defined in Definition 122 and Definition 124 to the validator, which the validator should be able to respond to.

Candidates are recovered by sending requests for specific indices of erasure encoded chunks (Section 13.1). A validator should request chunks by picking peers randomly and must recover at least $f + 1$ chunks, where $n = 3f + k$ and $k \in \{1, 2, 3\}$. n is the number of validators as specified in the session info, which can be fetched by the Runtime API as described in Section 26.6.11.

6.5. Approval Voting

The approval voting process ensures that only valid parachain blocks are finalized on the relay chain. After *backable* parachain candidates were submitted to the relay chain (Section 6.2.2), which can be retrieved via the Runtime API (Section 26.6.3), validators need to determine their assignments for each parachain and issue approvals for valid candidates, respectively disputes for invalid candidates. Since it cannot be expected that each validator verifies every single parachain candidate, this mechanism ensures that enough honest validators are selected to verify parachain candidates in order prevent the finalization of invalid blocks. If an honest validator detects an

invalid block which was approved by one or more validators, the honest validator must issue a disputes which wil cause escalations, resulting in consequences for all malicious parties, i.e. slashing. This mechanism is described more in Section 6.5.1.

6.5.1. Assignment Criteria

Validators determine their assignment based on a VRF mechanism, similar to the BABE consensus mechanism. First, validators generate an availability core VRF assignment (Definition 111), which indicates which availability core a validator is assigned to. Then a delayed availability core VRF assignment is generated which indicates at what point a validator should start the approval process. The delays are based on "tranches" (Section 6.5.2).

An assigned validator never broadcasts their assignment until relevant. Once the assigned validator is ready to check a candidate, the validator broadcasts their assignment by issuing an approval distribution message (Definition 119), where M is of variant *0*. Other assigned validators that receive that network message must keep track of if, expecting an approval vote following shortly after. Assigned validators can retrieve the candidate by using the availability recovery (Section 6.4.2) and then validate the candidate (Section 6.3).

The validator issues approval votes in form of a validator protocol message (Definition 114) respectively disputes (Section 6.6).

6.5.2. Tranches

Validators use a subjective, tick-based system to determine when the approval process should start. A validator starts the tick-based system when a new availability core candidates have been proposed, which can be retrieved via the Runtime API (Section 26.6.3) , and increments the tick every *500 milliseconds*. Each tick/increment is referred to as a "tranche", represented as an integer, starting at *0*.

As described in Section 6.5.1, the validator first executes the VRF mechanism to determine which parachains (availability cores) the validator is assigned to, then an additional VRF mechanism for each assigned parachain to determine the *delayed assignment*. The delayed assignment indicates the tranche at which the validator should start the approval process. A tranche of value *0* implies that the assignment should be started immediately, while later assignees of later tranches wait until it's their term to issue assignments, determined by their subjective, tick-based system.

Validators are required to track broadcasted assignments by other validators assigned to the same parachain, including verifying the VRF output. Once a valid assignment from a peer was received, the validator must wait for the following approval vote within a certain period as described in Section 26.6.11 by orienting itself on its local, tick-based system. If the waiting time after a broadcasted assignment exceeds the specified period, the validator interprets this behavior as a "no-show", indicating that more validators should commit on their tranche until enough approval votes have been collected.

If enough approval votes have been collected as described in Section 26.6.11, then assignees of later tranches do not have to start the approval process. Therefore, this tranche system serves as a mechanism to ensure that enough candidate approvals from a random set of validators are created without requiring all assigned validators to check the candidate.

Definition 110. Relay VRF Story

The relay VRF story is an array of random bytes derived from the VRF submitted within the block by the block author. The relay VRF story, T, is used as input to determine approval voting criteria and generated the following way:

$$T = \text{Transcript}(b_r, b_s, e_i, A)$$

where

- Transcript constructs a VRF transcript (Definition 150).
- b_r is the BABE randomness of the current epoch (Definition 79).
- b_s is the current BABE slot (Definition 65).
- e_i is the current BABE epoch index (Definition 65).
- A is the public key of the authority.

Definition 111. Availability Core VRF Assignment

An availability core VRF assignment is computed by a relay chain validator to determine which availability core (Section 6.8.6) a validator is assigned to and should vote for approvals. Computing this assignement relies on the VRF mechanism, transcripts and STROBE operations described further in Section 10.4.

The Runtime dictates how many assignments should be conducted by a validator, as specified in the session index which can be retrieved via the Runtime API (Section 26.6.11). The amount of assignments is referred to as "samples". For each iteration of the number of samples, the validator calculates an individual assignment, T, where the little-endian encoded sample number, s, is incremented by one. At the beginning of the iteration, S starts at value *0*.

The validator executes the following steps to retrieve a (possibly valid) core index:

$$t_1 \leftarrow \text{Transcript('A\&V MOD')} t_2 \leftarrow \text{append}(t_1, \text{'RC-VRF'}, R_s) t_3 \leftarrow \text{append}(t_2, \text{'sample'}, s) t_4 \leftarrow \text{append}(t_3, \text{'vrf-nm-pk'}, p_k) t_5 \leftarrow \text{meta-ad}(t_4, \text{'VRFHash'}, \text{False}) t_6 \leftarrow \text{meta-ad}(t_5, 64_{\text{le}}, \text{True}) i \leftarrow \text{prf}(t_6, \text{False}) o = s_k \cdot i$$

where s_k is the secret key, p_k is the public key and 64_{le} is the integer *64* encoded as little endian. R_s is the relay VRF story as defined in Definition 110. Following:

$$t_1 \leftarrow \text{Transcript('VRFResult')} t_2 \leftarrow \text{append}(t_1, \text{''}, \text{'A\&V CORE'}) t_3 \leftarrow \text{append}(t_2, \text{'vrf-in'}, i) t_4 \leftarrow \text{append}(t_3, \text{'vrf-out'}, o) t_5 \leftarrow \text{meta-ad}(t_4, \text{''}, \text{False}) t_6 \leftarrow \text{meta-ad}(t_5, 4_{\text{le}}, \text{True}) r \leftarrow \text{prf}(t_6, \text{False}) c_i = r \bmod a_c$$

where 4_{le} is the integer *4* encoded as little endian, r is the 4-byte challenge interpreted as a little endian encoded interger and a_c is the number of availability cores used during the active session, as defined in the session info retrieved by the Runtime API (Section 26.6.11). The resulting integer, c_i, indicates the parachain Id (Section 6.8.5). If the parachain Id doesn't exist, as can be retrieved by the Runtime API (Section 26.6.3), the validator discards that value and continues with the next iteration. If the Id does exist, the validators continues with the following steps:

$$t_1 \leftarrow \text{Transcript('A\&V ASSIGNED')} t_2 \leftarrow \text{append}(t_1, \text{'core'}, c_i) p \leftarrow \text{dleq\_prove}(t_2, i)$$

where dleq_prove is described in Definition 148. The resulting values of o, p and s are used to construct an assignment certificate (Definition 113) of kind *0*.

Definition 112. Delayed Availability Core VRF Assignment

The **delayed availability core VRF assignments** determined at what point a validator should start the approval process as described in Section 6.5.2. Computing this assignement relies on the VRF mechanism, transcripts and STROBE operations described further in Section 10.4.

The validator executes the following steps:

$t_1 \leftarrow \text{Transcript('A\&V DELAY')} t_2 \leftarrow \text{append}(t_1, \text{'RC-VRF'}, R_s) t_3 \leftarrow \text{append}(t_2, \text{'core'}, c_i) t_4 \leftarrow \text{append}(t_3, \text{'vrf-nm-pk'}, p_k) t_5 \leftarrow \text{meta-ad}(t_4, \text{'VRFHash'}, \text{False}) t_6 \leftarrow \text{meta-ad}(t_5, 64_{le}, \text{True}) i \leftarrow \text{prf}(t_6, \text{False}) o = s_k \cdot ip \leftarrow \text{dleq\_prove}(t_6, i)$

The resulting value p is the VRF proof (Definition 147). dleq_prove is described in Definition 148.

The tranche, d, is determined as:

$t_1 \leftarrow \text{Transcript('VRFResult')} t_2 \leftarrow \text{append}(t_1, \text{''}, \text{'A\&V TRANCHE'}) t_3 \leftarrow \text{append}(t_2, \text{'vrf-in'}, i) t_4 \leftarrow \text{append}(t_3, \text{'vrf-out'}, o) t_5 \leftarrow \text{meta-ad}(t_4, \text{''}, \text{False}) t_6 \leftarrow \text{meta-ad}(t_5, 4_{le}, \text{True}) c \leftarrow \text{prf}(t_6, \text{False}) d = d \bmod (d_c + d_z) - d_z$

where

- d_c is the number of delayed tranches by total as specified by the session info, retrieved via the Runtime API (Section 26.6.11).
- d_z is the zeroth delay tranche width as specified by the session info, retrieved via the Runtime API (Section 26.6.11)..

The resulting tranche, n, cannot be less than 0. If the tranche is less than 0, then $d = 0$. The resulting values o, p and c_i are used to construct an assignment certificate (<Definition 113) of kind *1*.

Definition 113. Assignment Certificate

The **Assignment Certificate** proves to the network that a Polkadot validator is assigned to an availability core and is therefore qualified for the approval of candidates, as clarified in Definition 111. This certificate contains the computed VRF output and is a datastructure of the following format:

$$(k, o, p)k = \begin{cases} 0 & -\&>; \quad s \\ 1 & -\&>; \quad c_i \end{cases}$$

where k indicates the kind of the certificate, respectively the value *0* proves the availability core assignment (Definition 111), followed by the sample number s, and the value *1* proves the delayed availability core assignment (Definition 112), followed by the core index c_i (Section 26.6.3). o is the VRF output and p is the VRF proof.

6.6. Disputes

Disputes are not documented yet.

6.7. Network Messages

The availability and validity process requires certain network messages to be exchanged between validators and collators.

6.7.1. Notification Messges

The notification messages are exchanged between validators, including messages sent by collators to validators. The protocol messages are exchanged based on a streaming notification substream (Section 4.5). The messages are SCALE encoded (Section 11.2).

Definition 114. Validator Protocol Message

The validator protocol message is a varying datatype used by validators to broadcast relevant information about certain steps in the A&V process. Specifically, this includes the backing process (Section 6.2) and the approval process (Section 6.5). The validator protocol message, M, is a varying datatype of the following format:

$$M = \begin{cases} 1 & -\&>; \ M_f \\ 3 & -\&>; \ M_s \\ 4 & -\&>; \ M_a \end{cases}$$

where

- M_f is a bitfield distribution message (Definition 118).
- M_s is a statement distribution message (Definition 117).
- M_a is a approval distribution message (Definition 119).

Definition 115. Collation Protocol Message

The collation protocol message, M, is a varying datatype of the following format:

$$M = \{(0, \ -\&>; \ , \ M_c)$$

where M_c is the collator message (Definition 116).

Definition 116. Collator Message

The collator message is sent as part of the collator protocol message (Definition 115). The collator message, M, is a varying datatype of the following format:

$$M = \begin{cases} 0 & -\&>; \quad (C_i, P_i, C_s) \\ 1 & -\&>; \quad H \\ 4 & -\&>; \quad (B_h, S) \end{cases}$$

where

- M is a varying datatype where *0* indicates the intent to advertise a collation and *1* indicates the advertisement of a collation to a validator. *4* indicates that a collation sent to a validator was seconded.
- C_i is the public key of the collator.
- P_i is the parachain Id (Section 6.8.5).
- C_s is the signature of the collator using the *PeerId* of the collators node.
- H is the hash of the parachain block (Section 6.8.3).
- S is a full statement (Definition 102).

Definition 117. Statement Distribution Message

The statement distribution message is sent as part of the validator protocol message (Definition 115) indicates the validity vote of a validator for a given candidate, described further in Section 6.2.1. The statement distribution message, M, is of varying type of the following format:

$$M = \begin{cases} 0 & -\&>; \quad (B_h, S) \\ 1 & -\&>; \quad S_m \end{cases} \quad S_m = (B_h, C_h, A_i, A_s)$$

where

- M is a varying datatype where *0* indicates a signed statement and *1* contains metadata about a seconded statement with a larger payload, such as a runtime upgrade. The candidate itself can be fetched via the request/response message (Definition 128).
- B_h is the hash of the relay chain parent, indicating the state this message is for.
- S is a full statement (Definition 102).
- A_i is the validator index in the authority set (Definition 63) that signed this message.
- A_s is the signature of the validator.

Definition 118. Bitfield Distribution Message

The bitfield distribution message is sent as part of the validator protocol message (Definition 114) and indicates the availability vote of a validator for a given candidate, described further in Section 6.4.1. This message is sent in form of a validator protocol message (Definition 114). The bitfield distribution message, M, is a datastructure of the following format:

$$M = \{(0, \ -\&>; , (B_h, \ P))P = (d, \ A_i, \ A_s)$$

where

- B_h is the hash of the relay chain parent, indicating the state this message is for.
- d is the bitfield array (Section 6.8.12).
- A_i is the validator index in the authority set (Definition 63) that signed this message.
- A_s is the signature of the validator.

Definition 119. Approval Distribution Message

The approval distribution message is sent as part of the validator protocol message (Definition 114) and indicates the approval vote of a validator for a given candidate, described further in Section 6.5.1. The approval distribution message, M, is a varying datatype of the following format:

$$M = \begin{cases} 0 \ -\&>; \ ((C, I)_0 \text{Unknown characterUnknown characterUnknown character}(C, I)_n) \\ 1 \ -\&>; \ (V_0, \text{ Unknown characterUnknown characterUnknown character} V_n) \end{cases} C = (B_h, \ A_i, \ c_a) c_a = (c_k, \ P_o, \ P_p) c_k = \begin{cases} 0 \text{Unknown characterUnknown characterUnknown characters} 1 \text{Unknown characterUnknown characterUnknown character} i \end{cases} V = (B_h, \ I, \ A_i, \ A_s)$$

where

- M is a varying datatype where *0* indicates assignments for candidates in recent, unfinalized blocks and *1* indicates approvals for candidates in some recent, unfinalized block.
- C is an assignment criterion which refers to the candidate under which the assignment is relevant by the block hash.
- I is an unsigned 32-bit integer indicating the index of the candidate, corresponding the the order of the availability cores (Section 26.6.3).
- B_h is the relay chain block hash where the candidate appears.
- A_i is the authority set Id (Definition 63) of the validator that created this message.
- A_s is the signature of the validator issuing this message.
- c_a is the certification of the assignment.
- c_k is a varying datatype where *0* indicates an assignment based on the VRF that authorized the relay chain block where the candidate was included, followed by a sample number, s. *1* indicates an assignment story based on the VRF that authorized the relay chain block where the candidate was included combined with the index of a particular core. This is described further in Section 6.5.
- P_o is a VRF output and P_p its corresponding proof.

6.7.2. Request & Response

The request & response network messages are sent and received between peers in the Polkadot network, including collators and non-validator nodes. Those messages are conducted on the request-response substreams (Section 4.5). The network messages are SCALE encoded as described in Section ?.

Definition 120. PoV Fetching Request

The PoV fetching request is sent by clients who want to retrieve a PoV block from a node. The request is a datastructure of the following format:

$$C_h$$

where C_h is the 256-bit hash of the PoV block. The response message is defined in Definition 121.

Definition 121. PoV Fetching Response

The PoV fetching response is sent by nodes to the clients who issued a PoV fetching request (Definition 120). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & -\&>; \ B \\ 1 & -\&>; \ \phi \end{cases}$$

where *0* is followed by the PoV block and *1* indicates that the PoV block was not found.

Definition 122. Chunk Fetching Request

The chunk fetching request is sent by clients who want to retrieve chunks of a parachain candidate. The request is a datastructure of the following format:

$$(C_h, i)$$

where C_h is the 256-bit hash of the parachain candidate and i is a 32-bit unsigned integer indicating the index of the chunk to fetch. The response message is defined in Definition 123.

Definition 123. Chunk Fetching Response

The chunk fetching response is sent by nodes to the clients who issued a chunk fetching request (Definition 122). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & -\&>; \ C_r \\ 1 & -\&>; \ \phi \end{cases} \quad C_r = (c, c_p)$$

where *0* is followed by the chunk response, C_r and *1* indicates that the requested chunk was not found. C_r contains the erasure-encoded chunk of data belonging to the candidate block, c, and c_p is that chunks proof in the Merkle tree. Both c and c_p are byte arrays of type $(b_n$Unknown characterUnknown characterUnknown character$b_m)$.

Definition 124. *Available Data Request*

The available data request is sent by clients who want to retrieve the PoV block of a parachain candidate. The request is a datastructure of the following format:

$$C_h$$

where C_h is the 256-bit candidate hash to get the available data for. The response message is defined in Definition 125.

Definition 125. *Available Data Response*

The available data response is sent by nodes to the clients who issued a available data request (Definition 124). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & -\&>; \quad A \\ 1 & -\&>; \quad \phi \end{cases} \quad A = (P_{ov}, D_{pv})$$

where *0* is followed by the available data, A, and *1* indicates the the requested candidate hash was not found. P_{ov} is the PoV block (Section 6.8.3) and D_{pv} is the persisted validation data (Definition 199).

Definition 126. *Collation Fetching Request*

The collation fetching request is sent by clients who want to retrieve the advertised collation at the specified relay chain block. The request is a datastructure of the following format:

$$(B_h, P_{id})$$

where B_h is the hash of the relay chain block and P_{id} is the parachain Id (Section 6.8.5). The response message is defined in Definition 127.

Definition 127. *Collation Fetching Response*

The collation fetching response is sent by nodes to the clients who issued a collation fetching request (Definition 126). The response, R, is a varying datatype of the following format:

$$R = \{(0, -\&>;, (C_r, B))$$

where 0 is followed by the candidate receipt (Definition 104), C_r, as and the PoV block (Section 6.8.3), B. This type does not notify the client about a statement that was not found.

Definition 128. Statement Fetching Request

The statement fetching request is sent by clients who want to retrieve statements about a given candidate. The request is a datastructure of the following format:

$$(B_h, C_h)$$

where B_h is the hash of the relay chain parent and C_h is the candidate hash that was used to create a committed candidate receipt (Definition 105). The response message is defined in Definition 129.

Definition 129. Statement Fetching Response

The statement fetching response is sent by nodes to the clients who issued a collation fetching request (Definition 128). The response, R, is a varying datatype of the following format:

$$R = \{(0, \ -\&>;, C_r)$$

where C_r is the committed candidate receipt (Definition 105). No response is returned if no statement is found.

6.7.2.1. Dispute Request

The dispute request is sent by clients who want to issue a dispute about a candidate. The request, D_r, is a datastructure of the following format:

$$D_r = (C_r, S_i, I_v, V_v) I_v = (A_i, A_s, k_i) V_v = (A_i, A_s, k_v) k_i = \{(0, \ -\&>;, \phi) k_v = \begin{cases} 0 & -\&>; \ \phi \\ 1 & -\&>; \ C_h \\ 2 & -\&>; \ C_h \\ 3 & -\&>; \ \phi \end{cases}$$

where

- C_r is the candidate that is being disputed. The structure is a candidate receipt (Definition 104).
- S_i is an unsigned 32-bit integer indicating the session index the candidate appears in.
- I_v is the invalid vote that makes up the request.
- V_v is the valid vote that makes this dispute request valid.
- A_i is an unsigned 32-bit integer indicating the validator index in the authority set (Definition 63).
- A_s is the signature of the validator.
- k_i is a varying datatype and implies the dispute statement. *0* indicates an explicit statement.
- k_v is a varying datatype and implies the dispute statement.
  - 0 indicates an explicit statement.
  - 1 indicates a seconded statement on a candidate, C_h, from the backing phase. C_h is the hash of the candidate.
  - 2 indicates a valid statement on a candidate, C_h, from the backing phase. C_h is the hash of the

candidate.

- 3 indicates an approval vote from the approval checking phase.

The response message is defined in Section 6.7.2.2.

6.7.2.2. Dispute Response

The dispute response is sent by nodes to the clients who who issued a dispute request (Section 6.7.2.1). The response, R, is a varying type of the following format:

$$R = \{(0, -\&>;, \phi)$$

where 0 indicates that the dispute was successfully processed.

6.8. Definitions

6.8.1. Collator

A collator is a parachain node that sends parachain blocks, known as candidates (Section 6.8.2), to the relay chain validators. The relay chain validators are not concerned how the collator works or how it creates candidates.

6.8.2. Candidate

A candidate is a submitted parachain block (Section 6.8.3) to the relay chain validators. A parachain block stops being referred to as a candidate as soon it has been finalized.

6.8.3. Parachain Block

A parachain block or a Proof-of-Validity block (PoV block) contains the necessary data to for parachain specific state transition logic. Relay chain validators are not concerned with the inner structure of the block and treat it as a byte array.

6.8.4. Head Data

The head data is contains information about a parachain block (Section 6.8.3). The head data is returned by executing the parachain Runtime and relay chain validators are not concerned with its inner structure and treat it as a byte arrays.

6.8.5. Parachain Id

The Parachain Id is a unique, unsigned 32-bit integer which serves as an identifier of a parachain, assigned by the Runtime.

6.8.6. Availability Core

Availability cores are slots used to process parachains. The Runtime assigns each parachain to a availability core and validators can fetch information about the cores, such as parachain block candidates, by calling the appropriate Runtime API (Section 26.6.3). Validators are not concerned

with the internal workings from the Runtimes perspective.

6.8.7. Validator Groups

Validator groups indicate which validators are responsible for creating backable candidates for parachains (Section 6.2), and are assigned by the Runtime (Section 26.6.2). Validators are not concerned with the internal workings from the Runtimes perspective. Collators can use this information for submitting blocks.

6.8.8. Upward Message

An upward message is an opaque byte array sent from a parachain to a relay chain.

6.8.9. Downward Message

A downward message is an opaque byte array received by the parachain from the relay chain.

6.8.10. Outbound HRMP Message

An outbound HRMP message (Horizontal Relay-routed Message Passing) is sent from the perspective of a sender of a parachain to an other parachain by passing it through the relay chain. It's a datastructure of the following format:

$$(I, M)$$

where I is the recipient Id (Section 6.8.5) and M is an upward message (Section 6.8.8).

6.8.11. Inbound HRMP Message

An inbound HRMP message (Horizontal Relay-routed Message Passing) is seen from the perspective of a recipient parachain sent from an other parachain by passing it through the relay chain. It's a datastructure of the following format:

$$(N, M)$$

where N is the unsigned 32-bit integer indicating the relay chain block number at which the message was passed down to the recipient parachain and M is a downward message (Section 6.8.9).

6.8.12. Bitfield Array

A bitfield array contains single-bit values which indidate whether a candidate is available. The number of items is equal of to the number of availability cores (Section 6.8.6) and each bit represents a vote on the corresponding core in the given order. Respectively, if the single bit equals 1, then the Polkadot validator claims that the availability core is occupied, there exists a committed candidate receipt (Definition 105) and that the validator has a stored chunk of the parachain block (Section 6.8.3).

Runtime Specification

Description of various useful Runtime internals

Chapter 7. Extrinsics

7.1. Introduction

An extrinsic is a SCALE encoded array consisting of a version number, signature, and varying data types indicating the resulting Runtime function to be called, including the parameters required for that function to be executed.

7.2. Preliminaries

Definition 130. Extrinsic

An extrinsic , tx, is a tuple consisting of the extrinsic version, T_v (Definition 131), and the body of the extrinsic, T_b.

$$tx := (T_v, T_b)$$

The value of T_b varies for each version. The current version 4 is described in Section 7.3.1.

Definition 131. Extrinsic Version

T_v is a 8-bit bitfield and defines the extrinsic version. The required format of an extrinsic body, T_b, is dictated by the Runtime. Older or unsupported version are rejected.

The first bit of T_v indicates whether the transaction is **signed** (1) or **unsigned** (0). The remaining 7-bits represent the version number. As an example, for extrinsic format version 4, an signed extrinsic represents T_v as 132 while a unsigned extrinsic represents it as 4.

7.3. Extrinsics Body

7.3.1. Version 4

Version 4 of the Polkadot extrinsic format is defined as follows:

$$T_b := (A_i, Sig, E, M_i, F_i(m))$$

where each values represents:

- A_i: the 32-byte address of the sender (Definition 132).
- Sig: the signature of the sender (Definition 133).
- E: the extra data for the extrinsic (Definition 134).
- M_i: the indicator of the Polkadot module (Definition 135).
- $F_i(m)$: the indicator of the function of the Polkadot module (Definition 136).

Definition 132. Extrinsic Address

Account Id, A_i, is the 32-byte address of the sender of the extrinsic as described in the external SS58 address format.

Definition 133. Extrinsic Signature

The signature, Sig, is a varying data type indicating the used signature type, followed by the signature created by the extrinsic author. The following types are supported:

$$Sig := \begin{cases} 0, & \text{Ed25519, followed by:}(b_0, ..., b_{63}) \\ 1, & \text{Sr25519, followed by:}(b_0, ..., b_{63}) \\ 2, & \text{Ecdsa, followed by:}(b_0, ..., b_{64}) \end{cases}$$

Signature types vary in sizes, but each individual type is always fixed-size and therefore does not contain a length prefix. `Ed25519` and `Sr25519` signatures are 512-bit while `Ecdsa` is 520-bit, where the last 8 bits are the recovery ID.

The signature is created by signing payload P.

$$P := \begin{cases} Raw, & \text{if} |Raw| \leq 256 \\ Blake2(Raw), & \text{if} |Raw| > 256 \end{cases}$$
$$Raw := (M_i, F_i(m), E, R_v, F_v, H_h(G), H_h(B))$$

where each value represents:

- M_i: the module indicator (Definition 135).
- $F_i(m)$: the function indicator of the module (Definition 136).
- E: the extra data (Definition 134).
- R_v: a UINT32 containing the specification version of 14.
- F_v: a UINT32 containing the format version of 2.
- $H_h(G)$: a 32-byte array containing the genesis hash.
- $H_h(B)$: a 32-byte array containing the hash of the block which starts the mortality period, as described in Definition 137.

Definition 134. Extra Data

Extra data, E, is a tuple containing additional meta data about the extrinsic and the system it is meant to be executed in.

$$E := (T_{mor}, N, P_t)$$

where each value represents:

- T_{mor}: contains the SCALE encoded mortality of the extrinsic (Definition 137).
- N: a compact integer containing the nonce of the sender. The nonce must be incremented by one for each extrinsic created, otherwise the Polkadot network will reject the extrinsic.
- P_t: a compact integer containing the transactor pay including tip.

Definition 135. Module Indicator

M_i is an indicator for the Runtime to which Polkadot *module*, m, the extrinsic should be forwarded to.

M_i is a varying data type pointing to every module exposed to the network.

$$M_i := \begin{cases} 0, & \text{System} \\ 1, & \text{Utility} \\ \dots & \\ 7, & \text{Balances} \\ \dots & \end{cases}$$

Definition 136. Function Indicator

$F_i(m)$ is a tuple which contains an indicator, m_i, for the Runtime to which *function* within the Polkadot *module*, m, the extrinsic should be forwarded to. This indicator is followed by the concatenated and SCALE encoded parameters of the corresponding function, $params$.

$$F_i(m) := (m_i, params)$$

The value of m_i varies for each Polkadot module, since every module offers different functions. As an example, the `Balances` module has the following functions:

$$Balances_i := \begin{cases} 0, & \text{transfer} \\ 1, & \text{set\_balance} \\ 2, & \text{force\_transfer} \\ 3, & \text{transfer\_keep\_alive} \\ \dots & \end{cases}$$

7.3.2. Mortality

Definition 137. Extrinsic Mortality

Extrinsic **mortality** is a mechanism which ensures that an extrinsic is only valid within a certain period of the ongoing Polkadot lifetime. Extrinsics can also be immortal, as clarified in Section 7.3.2.2.

The mortality mechanism works with two related values:

- M_{per}: the period of validity in terms of block numbers from the block hash specified as $H_h(B)$ in the payload (Definition 133). The requirement is $M_{per} \geq 4$ and M_{per} must be the power of two, such as 32, 64, 128, etc.
- M_{pha}: the phase in the period that this extrinsic's lifetime begins. This value is calculated with a formula and validators can use this value in order to determine which block hash is included in the payload. The requirement is $M_{pha} < M_{per}$.

In order to tie a transaction's lifetime to a certain block ($H_i(B)$) after it was issued, without wasting precious space for block hashes, block numbers are divided into regular periods and the lifetime is instead expressed as a "phase" (M_{pha}) from these regular boundaries:

$$M_{pha} = H_i(B) \, mod \, M_{per}$$

M_{per} and M_{pha} are then included in the extrinsic, as clarified in Definition 134, in the SCALE encoded form of T_{mor} (Section 7.3.2.2). Polkadot validators can use M_{pha} to figure out the block hash included in the payload, which will therefore result in a valid signature if the extrinsic is within the specified period or an invalid signature if the extrinsic "died".

7.3.2.1. Example

The extrinsic author choses $M_{per} = 256$ at block 10'000, resulting with $M_{pha} = 16$. The extrinsic is then valid for blocks ranging from 10'000 to 10'256.

7.3.2.2. Encoding

T_{mor} refers to the SCALE encoded form of type M_{per} and M_{pha}. T_{mor} is the size of two bytes if the extrinsic is considered mortal, or simply one bytes with the value equal to zero if the extrinsic is considered immortal.

$$T_{mor} := Enc_{SC}(M_{per}, M_{pha})$$

The SCALE encoded representation of mortality T_{mor} deviates from most other types, as it's specialized to be the smallest possible value, as described in Algorithm 23 and Algorithm 24.

If the extrinsic is immortal, specify a single byte with the value equal to zero.

Algorithm 23 Encode Mortality

Require: M_{per}, M_{pha}

return 0 **if** *extrinsic is immortal*
init factor = Limit($M_{per} >> 12, 1, \phi$)
init left = Limit$_{M_{pha}}$(TZ(M_{per}) − 1, 1, 15)
init right = factor << 4
return left ∣ right

Algorithm 23. Encode Mortality

Algorithm 24 Decode Mortality

Require: T_{mor}

return *Immortal* **if** $T^{b0}_{mor} = 0$
init enc = $T^{b0}_{mor} + (T^{b1}_{mor} << 8)$
init M_{per} = 2 << (enc mod (1 << 4))
init factor = Limit($M_{per} >> 12, 1, \phi$)
init M_{pha} = (enc >> 4) ∣ factor
return (M_{per}, M_{pha})

Algorithm 24. Decode Mortality

- T^{b0}_{mor}: the first byte of T_{mor}.
- T^{b1}_{mor}: the second byte of T_{mor}.
- Limit(num, min, max): Ensures that num is between min and max. If min or max is defined as ϕ, then there is no requirement for the specified minimum/maximum.
- TZ(num): returns the number of trailing zeros in the binary representation of num. For example, the binary representation of `40` is `0010 1000`, which has three trailing zeros.
- >>: performs a binary right shift operation.
- <<: performs a binary left shift operation.
- ∣: performs a bitwise OR operation.

Chapter 8. Weights

8.1. Motivation

The Polkadot network, like any other permissionless system, needs to implement a mechanism to measure and to limit the usage in order to establish an economic incentive structure, to prevent the network overload, and to mitigate DoS vulnerabilities. In particular, Polkadot enforces a limited time-window for block producers to create a block, including limitations on block size, which can make the selection and execution of certain extrinsics too expensive and decelerate the network.

In contrast to some other systems such as Ethereum which implement fine measurement for each executed low-level operation by smart contracts, known as gas metering, Polkadot takes a more relaxed approach by implementing a measuring system where the cost of the transactions (referred to as 'extrinsics') are determined before execution and are known as the weight system.

The Polkadot weight system introduces a mechanism for block producers to measure the cost of running the extrinsics and determine how "heavy" it is in terms of execution time. Within this mechanism, block producers can select a set of extrinsics and saturate the block to its fullest potential without exceeding any limitations (as described in Section 8.2.1). Moreover, the weight system can be used to calculate a fee for executing each extrinsics according to its weight (as described in Section 8.6.1).

Additionally, Polkadot introduces a specified block ratio (as defined in Section 8.2.1), ensuring that only a certain portion of the total block size gets used for regular extrinsics. The remaining space is reserved for critical, operational extrinsics required for the functionality by Polkadot itself.

To begin, we introduce in Section 8.2 the assumption upon which the Polkadot transaction weight system is designed. In Section 8.2.1, we discuss the limitation Polkadot needs to enforce on the block size. In Section 8.3, we describe in detail the procedure upon which the weight of any transaction should be calculated. In Section 8.5, we present how we apply this procedure to compute the weight of particular runtime functions.

8.2. Assumptions

In this section, we define the concept of weight and we discuss the considerations that need to be accounted for when assigning weight to transactions. These considerations are essential in order for the weight system to deliver its fundamental mission, i.e. the fair distribution of network resources and preventing a network overload. In this regard, weights serve as an indicator on whether a block is considered full and how much space is left for remaining, pending extrinsics. Extrinsics which require too many resources are discarded. More formally, the weight system should:

- prevent the block from being filled with too many extrinsics
- avoid extrinsics where its execution takes too long, by assigning a transaction fee to each extrinsic proportional to their resource consumption.

These concepts are formalized in Definition 138 and Definition 141:

Definition 138. Block Length

For a block B with $Head(B)$ and $Body(B)$ the block length of B, $Len(B)$, is defined as the amount of raw bytes of B.

Definition 139. Target Time per Block

Targeted time per block denoted by $T(B)$ implies the amount of seconds that a new block should be produced by a validator. The transaction weights must consider $T(B)$ in order to set restrictions on time intensive transactions in order to saturate the block to its fullest potential until $T(B)$ is reached.

Definition 140. Block Target Time

Available block ration reserved for normal, noted by $R(B)$, is defined as the maximum weight of none-operational transactions in the Body of B divided by $Len(B)$.

Definition 141. Block Limits

Polkadot block limits as defined here should be respected by each block producer for the produced block B to be deemed valid:

- $Len(B) \leq 5 \times 1\,'\,024 \times 1\,'\,024 = 5\,'\,242\,'\,880$ Bytes
- $T(B) = 6 seconds$
- $R(B) \leq 0.75$

Definition 142. Weight Function

The Polkadot transaction weight function denoted by $\mathcal{W}$ as follows:

$$\mathcal{W} : \mathcal{E} \rightarrow \mathbb{N}$$
$$\mathcal{W} : E \mapsto w$$

where w is a non-negative integer representing the weight of the extrinsic E. We define the weight of all inherent extrinsics as defined in the Section 3.2.3 to be equal to 0. We extend the definition of $\mathcal{W}$ function to compute the weight of the block as sum of weight of all extrinsics it includes:

$$\mathcal{W} : \mathcal{B} \rightarrow \mathbb{N}$$
$$\mathcal{W} : B \mapsto \sum_{E \in B} (W(E))$$

In the remainder of this section, we discuss the requirements to which the weight function needs to comply to.

- Computations of function $\mathcal{W}(E)$ must be determined before execution of that E.

- Due to the limited time window, computations of $\mathcal{W}$ must be done quickly and consume few resources themselves.
- $\mathcal{W}$ must be self contained and must not require I/O on the chain state. $\mathcal{W}(E)$ must depend solely on the Runtime function representing E and its parameters.

Heuristically, "heaviness" corresponds to the execution time of an extrinsic. In that way, the $\mathcal{W}$ value for various extrinsics should be proportional to their execution time. For example, if Extrinsic A takes three times longer to execute than Extrinsic B, then Extrinsic A should roughly weighs 3 times of Extrinsic B. Or:

$$\mathcal{W}(A) \approx 3 \times \mathcal{W}(B)$$

Nonetheless, $\mathcal{W}(E)$ can be manipulated depending on the priority of E the chain is supposed to endorse.

8.2.1. Limitations

In this section we discuss how applying the limitation defined in Definition 141 can be translated to limitation $\mathcal{W}$. In order to be able to translate those into concrete numbers, we need to identify an arbitrary maximum weight to which we scale all other computations. For that we first define the block weight and then assume a maximum on it block length in Definition 143:

Definition 143. Block Weight

We define the block weight of block B, formally denoted as $\mathcal{W}(B)$, to be:

$$\mathcal{W}(B) = \sum_{n=0}^{|\mathcal{E}|} (W(E_n))$$

We require that:

$$\mathcal{W}(B) < 2 \prime 000 \prime 000 \prime 000 \prime 000$$

The weights must fulfill the requirements as noted by the fundamentals and limitations, and can be assigned as the author sees fit. As a simple example, consider a maximum block weight of 1’000’000’000, an available ratio of 75% and a targeted transaction throughput of 500 transactions, we could assign the (average) weight for each transaction at about 1’500’000. Block producers have economic incentive to include as many extrinsics as possible (without exceeding limitations) into a block before reaching the targeted block time. Weights give indicators to block producers on which extrinsics to include in order to reach the blocks fullest potential.

8.3. Calculation of the weight function

In order to calculate weight of block B, $\mathcal{W}(B)$, one needs to evaluate the weight of each transaction included in the block. Each transaction causes the execution certain Runtime functions. As such, to calculate the weight of a transaction, those functions must be analyzed in order to determine parts of the code which can significantly contribute to the execution time and consume resources such as loops, I/O operations, and data manipulation. Subsequently the performance and execution time of each part will be evaluated based on variety of input parameters. Based on those observations,

weights are assigned Runtime functions or parameters which contribute to long execution times. These sub component of the code are discussed in Section 8.4.1.

The general algorithm to calculate $\mathcal{W}(E)$ is described in the Section 8.4.

8.4. Benchmarking

Calculating the extrinsic weight solely based on theoretical complexity of the underlying implementation proves to be too complicated and unreliable at the same time. Certain decisions in the source code architecture, internal communication within the Runtime or other design choices could add enough overhead to make the asymptotic complexity practically meaningless.

On the other hand, benchmarking an extrinsics in a black-box fashion could (using random parameters) most centainly results in missing corner cases and worst case scenarios. Instead, we benchmark all available Runtime functions which are invoked in the course of execution of extrinsics with a large collection of carefully selected input parameters and use the result of the benchmarking process to evaluate $\mathcal{W}(E)$.

In order to select useful parameters, the Runtime functions have to be analyzed to fully understand which behaviors or conditions can result in expensive execution times, which is described closer in Section 8.4.1. Not every possible benchmarking outcome can be invoked by varying input parameters of the Runtime function. In some circumstances, preliminary work is required before a specific benchmark can be reliably measured, such as creating certain preexisting entries in the storage or other changes to the environment.

The Practical Examples (Section 8.5) covers the analysis process and the implementation of preliminary work in more detail.

8.4.1. Primitive Types

The Runtime reuses components, known as "primitives", to interact with the state storage. The execution cost of those primitives can be measured and a weight should be applied for each occurrence within the Runtime code.

For storage, Polkadot uses three different types of storage types across its modules, depending on the context:

- **Value**: Operations on a single value. The final key-value pair is stored under the key:

```
hash(module_prefix) + hash(storage_prefix)
```

- **Map**: Operations on multiple values, datasets, where each entry has its corresponding, unique key. The final key-value pair is stored under the key:

```
hash(module_prefix) + hash(storage_prefix) + hash(encode(key))
```

- **Double map**: Just like **Map**, but uses two keys instead of one. This type is also known as "child

storage", where the first key is the "parent key" and the second key is the "child key". This is useful in order to scope storage entries (child keys) under a certain `context` (parent key), which is arbitrary. Therefore, one can have separated storage entries based on the context. The final key-value pair is stored under the key:

```
hash(module_prefix) + hash(storage_prefix)
  + hash(encode(key1)) + hash(encode(key2))
```

It depends on the functionality of the Runtime module (or its sub-processes, rather) which storage type to use. In some cases, only a single value is required. In others, multiple values need to be fetched or inserted from/into the database.

Those lower level types get abstracted over in each individual Runtime module using the `decl_storage!` macro. Therefore, each module specifies its own types that are used as input and output values. The abstractions do give indicators on what operations must be closely observed and where potential performance penalties and attack vectors are possible.

8.4.1.1. Considerations

The storage layout is mostly the same for every primitive type, primarily differentiated by using special prefixes for the storage key. Big differences arise on how the primitive types are used in the Runtime function, on whether single values or entire datasets are being worked on. Single value operations are generally quite cheap and its execution time does not vary depending on the data that's being processed. However, excessive overhead can appear when I/O operations are executed repeatedly, such as in loops. Especially, when the amount of loop iterations can be influenced by the caller of the function or by certain conditions in the state storage.

Maps, in contrast, have additional overhead when inserting or retrieving datasets, which vary in sizes. Additionally, the Runtime function has to process each item inside that list.

Indicators for performance penalties:

- **Fixed iterations and datasets** - Fixed iterations and datasets can increase the overall cost of the Runtime functions, but the execution time does not vary depending on the input parameters or storage entries. A base Weight is appropriate in this case.
- **Adjustable iterations and datasets** - If the amount of iterations or datasets depend on the input parameters of the caller or specific entries in storage, then a certain weight should be applied for each (additional) iteration or item. The Runtime defines the maximum value for such cases. If it doesn't, it unconditionally has to and the Runtime module must be adjusted. When selecting parameters for benchmarking, the benchmarks should range from the minimum value to the maximum value, as described in Definition 144.
- **Input parameters** - Input parameters that users pass on to the Runtime function can result in expensive operations. Depending on the data type, it can be appropriate to add additional weights based on certain properties, such as data size, assuming the data type allows varying sizes. The Runtime must define limits on those properties. If it doesn't, it unconditionally has to and the Runtime module must be adjusted. When selecting parameters for benchmarking, the benchmarks should range from the minimum values to the maximum value, as described in

paragraph Definition 144.

Definition 144. Maximum Value

> What the maximum value should be really depends on the functionality that the Runtime function is trying to provide. If the choice for that value is not obvious, then it's advised to run benchmarks on a big range of values and pick a conservative value below the `targeted time per block` limit as described in section Section 8.2.1.

8.4.2. Parameters

The inputs parameters highly vary depending on the Runtime function and must therefore be carefully selected. The benchmarks should use input parameters which will most likely be used in regular cases, as intended by the authors, but must also consider worst case scenarios and inputs which might decelerate or heavily impact performance of the function. The input parameters should be randomized in order to cause various effects in behaviors on certain values, such as memory relocations and other outcomes that can impact performance.

It's not possible to benchmark every single value. However, one should select a range of inputs to benchmark, spanning from the minimum value to the maximum value which will most likely exceed the expected usage of that function. This is described in more detail in Section 8.4.1.1. The benchmarks should run individual executions/iterations within that range, where the chosen parameters should give insight on the execution time. Selecting imprecise parameters or too extreme ranges might indicate an inaccurate result of the function as it will be used in production. Therefore, when a range of input parameters gets benchmarked, the result of each individual parameter should be recorded and optionally visualized, then the necessary adjustment can be made. Generally, the worst case scenario should be assigned as the weight value for the corresponding runtime function.

Additionally, given the distinction theoretical and practical usage, the author reserves the right to make adjustments to the input parameters and assigned weights according to the observed behavior of the actual, real-world network.

8.4.2.1. Weight Refunds

When assigning the final weight, the worst case scenario of each runtime function should be used. The runtime can then additional "refund" the amount of weights which were overestimated once the runtime function is actually executed.

The Polkadot runtime only returns weights if the difference between the assigned weight and the actual weight calculated during execution is greater than 20%.

8.4.3. Storage I/O cost

It is advised to benchmark the raw I/O operations of the database and assign "base weights" for each I/O operation type, such as insertion, deletion, querying, etc. When a runtime function is executed, the runtime can then add those base weights of each used operation in order to calculate the final weight.

8.4.4. Environment

The benchmarks should be executed on clean systems without interference of other processes or software. Additionally, the benchmarks should be executed on multiple machines with different system resources, such as CPU performance, CPU cores, RAM and storage speed.

8.5. Practical examples

This section walks through Runtime functions available in the Polkadot Runtime to demonstrate the analysis process as described in Section 8.4.1.

In order for certain benchmarks to produce conditions where resource heavy computation or excessive I/O can be observed, the benchmarks might require some preliminary work on the environment, since those conditions cannot be created with simply selected parameters. The analysis process shows indicators on how the preliminary work should be implemented.

8.5.1. Practical Example #1: `request_judgement`

In Polkadot, accounts can save information about themselves on-chain, known as the "Identity Info". This includes information such as display name, legal name, email address and so on. Polkadot offers a set of trusted registrars, entities elected by a Polkadot public referendum, which can verify the specified contact addresses of the identities, such as Email, and vouch on whether the identity actually owns those accounts. This can be achieved, for example, by sending a challenge to the specified address and requesting a signature as a response. The verification is done off-chain, while the final judgement is saved onchain, directly in the corresponding Identity Info. It's also note worthy that Identity Info can contain additional fields, set manually by the corresponding account holder.

Information such as legal name must be verified by ID card or passport submission.

The function `request_judgement` from the `identity` pallet allows users to request judgement from a specific registrar.

```
(func $request_judgement (param $req_index int) (param $max_fee int))
```

- `req_index`: the index which is assigned to the registrar.
- `max_fee`: the maximum fee the requester is willing to pay. The judgement fee varies for each registrar.

Studying this function reveals multiple design choices that can impact performance, as it will be revealed by this analysis.

8.5.1.1. Analysis

First, it fetches a list of current registrars from storage and then searches that list for the specified registrar index.

```
let registrars = <Registrars<T>>::get();
let registrar = registrars.get(reg_index as usize).and_then(Option::as_ref)
  .ok_or(Error::<T>::EmptyIndex)?;
```

Then, it searches for the Identity Info from storage, based on the sender of the transaction.

```
let mut id = <IdentityOf<T>>::get(&sender).ok_or(Error::<T>::NoIdentity)?;
```

The Identity Info contains all fields that have a data in them, set by the corresponding owner of the identity, in an ordered form. It then proceeds to search for the specific field type that will be inserted or updated, such as email address. If the entry can be found, the corresponding value is to the value passed on as the function parameters (assuming the registrar is not "stickied", which implies it cannot be changed). If the entry cannot be found, the value is inserted into the index where a matching element can be inserted while maintaining sorted order. This results in memory reallocation, which increases resource consumption.

```
match id.judgements.binary_search_by_key(&reg_index, |x| x.0) {
  Ok(i) => if id.judgements[i].1.is_sticky() {
    Err(Error::<T>::StickyJudgement)?
  } else {
    id.judgements[i] = item
  },
  Err(i) => id.judgements.insert(i, item),
}
```

In the end, the function deposits the specified `max_fee` balance, which can later be redeemed by the registrar. Then, an event is created to insert the Identity Info into storage. The creation of events is lightweight, but its execution is what will actually commit the state changes.

```
T::Currency::reserve(&sender, registrar.fee)?;
<IdentityOf<T>>::insert(&sender, id);
Self::deposit_event(RawEvent::JudgementRequested(sender, reg_index));
```

8.5.1.2. Considerations

The following points must be considered:

- Varying count of registrars.
- Varying count of preexisting accounts in storage.
- The specified registrar is searched for in the Identity Info. An identity can be judged by as many registrars as the identity owner issues requests for, therefore increase its footprint in the state storage. Additionally, if a new value gets inserted into the byte array, memory get reallocated. Depending on the size of the Identity Info, the execution time can vary.
- The Identity Info can contain only a few fields or many. It is legitimate to introduce additional

weights for changes the owner/sender has influence over, such as the additional fields in the Identity Info.

8.5.1.3. Benchmarking Framework

The Polkadot Runtime specifies the `MaxRegistrars` constant, which will prevent the list of registrars of reaching an undesired length. This value should have some influence on the benchmarking process.

The benchmarking implementation of for the function *request_judgement* can be defined as follows:

Algorithm 25 "request_judgement"Runtime function benchmark
Ensure: W
1: **init** collection = {}
2: **for** amount ← 1, MaxRegistrars **do**
3: Generate-Registrars(amount)
4: caller ← Create-Account("caller", 1)
5: Set-Balance(caller, 100)
6: time ← Timer(Request-Judgement(Random(amount), 100))
7: Add-To(collection, time)
8: **end for**
9: W ← Compute-Weight(collection)
10: **return** W

Algorithm 25. "request_judgement" ` Runtime function benchmark

where

- Generate-Registrars(*amount*)

  Creates number of registrars and inserts those records into storage.

- Create-Account(*name*, *index*)

  Creates a Blake2 hash of the concatenated input of name and index represent- ing the address of a account. This function only creates an address and does not conduct any I/O.

- Set-Balance(*account*, *balance*)

  Sets a initial balance for the specified account in the storage state.

- Timer(*function*)

  Measures the time from the start of the specified f unction to its completion.

- Request-Judgement(*registrar_index*, *max_fee*)

  Calls the corresponding request_judgement Runtime function and passes on the required parameters.

- Random(*num*)

  Picks a random number between 0 and num. This should be used when the benchmark should account for unpredictable values.

- Add-To(*collection*, *time*)

  Adds a returned time measurement (time) to collection.

- Compute-Weight(*collection*)

  Computes the resulting weight based on the time measurements in the collection. The worst case scenario should be chosen (the highest value).

8.5.2. Practical Example #2: `payout_stakers`

8.5.2.1. Analysis

The function `payout_stakers` from the `staking` Pallet can be called by a single account in order to payout the reward for all nominators who back a particular validator. The reward also covers the validator's share. This function is interesting because it iterates over a range of nominators, which varies, and does I/O operation for each of them.

First, this function makes few basic checks to verify if the specified era is not higher then the current era (as it is not in the future) and is within the allowed range also known as "history depth", as specified by the Runtime. After that, it fetches the era payout from storage and additionally verifies whether the specified account is indeed a validator and receives the corresponding "Ledger". The Ledger keeps information about the stash key, controller key and other informatin such as actively bonded balance and a list of tracked rewards. The function only retains the entries of the history depth, and conducts a binary search for the specified era.

```
let era_payout = <ErasValidatorReward<T>>::get(&era)
  .ok_or_else(|| Error::<T>::InvalidEraToReward)?;

let controller = Self::bonded(&validator_stash).ok_or(Error::<T>::NotStash)?;
let mut ledger = <Ledger<T>>::get(&controller).ok_or_else(|| Error::<T>::
NotController)?;
```

```
ledger.claimed_rewards.retain(|&x| x >= current_era.saturating_sub(history_depth));
match ledger.claimed_rewards.binary_search(&era) {
  Ok(_) => Err(Error::<T>::AlreadyClaimed)?,
  Err(pos) => ledger.claimed_rewards.insert(pos, era),
}
```

The retained claimed rewards are inserted back into storage.

```
<Ledger<T>>::insert(&controller, &ledger);
```

As an optimization, Runtime only fetches a list of the 64 highest staked nominators, although this might be changed in the future. Accordingly, any lower staked nominator gets no reward.

```
let exposure = <ErasStakersClipped<T>>::get(&era, &ledger.stash);
```

Next, the function gets the era reward points from storage.

```
let era_reward_points = <ErasRewardPoints<T>>::get(&era);
```

After that, the payout is split among the validator and its nominators. The validators receives the

payment first, creating an insertion into storage and sending a deposit event to the scheduler.

```
if let Some(imbalance) = Self::make_payout(
  &ledger.stash,
  validator_staking_payout + validator_commission_payout
) {
  Self::deposit_event(RawEvent::Reward(ledger.stash, imbalance.peek()));
}
```

Then, the nominators receive their payout rewards. The functions loops over the nominator list, conducting an insertion into storage and a creation of a deposit event for each of the nominators.

```
for nominator in exposure.others.iter() {
  let nominator_exposure_part = Perbill::from_rational_approximation(
    nominator.value,
    exposure.total,
  );

  let nominator_reward: BalanceOf<T> = nominator_exposure_part *
validator_leftover_payout;
  // We can now make nominator payout:
  if let Some(imbalance) = Self::make_payout(&nominator.who, nominator_reward) {
    Self::deposit_event(RawEvent::Reward(nominator.who.clone(), imbalance.peek()));
  }
}
```

8.5.2.2. Considerations

The following points must be considered:

- The Ledger contains a varying list of claimed rewards. Fetching, retaining and searching through it can affect execution time. The retained list is inserted back into storage.
- Looping through a list of nominators and creating I/O operations for each increases execution time. The Runtime fetches up to 64 nominators.

8.5.2.3. Benchmarking Framework

Definition 145. History Depth

H□istory Depth indicated as `MaxNominatorRewardedPerValidator` is a fixed constant specified by the Polkadot Runtime which dictates the number of Eras the Runtime will reward nominators and validators for.

Definition 146. Maximum Nominator Reward

> M□aximum Nominator Rewarded Per Validator indicated as `MaxNominatorRewardedPerValidator`, specifies the maximum amount of the highest-staked nominators which will get a reward. Those values should have some influence in the benchmarking process.

The benchmarking implementation for the function $payout\_stakers$ can be defined as follows:

Algorithm 26 "payout_stakers" Runtime function benchmark

Ensure: W

```
init collection = {}
for amount ← 1, MaxNominatorRewardedPerValidator do
    for era_depth ← 1, HistoryDepth do
        validator ← Generate-Validator()
        Validate(validator)
        nominators ← Generate-Nominators(amount)
        for nominator ∈ nominators do
            Nominate(validator, nominator)
        end for
        era_index ← Create-Rewards(validator, nominators, era_depth)
        time ← Timer(Payout-Stakers(validator), era_index)
        Add-To(collection, time)
    end for
end for
W ← Compute-Weight(collection)
return W
```

Algorithm 26. "payout_stakers" Runtime function benchmark

where

- Generate-Validator()

  Creates a validators with some unbonded balances.

- Validate(*validator*)

  Bonds balances of validator and bonds balances.

- Generate-Nominators(*amount*)

  Creates the amount of nominators with some unbonded balances.

- Nominate(*validator*, *nominator*)

  Starts nomination of nominator for validator by bonding balances.

- Create-Rewards(*validator*, *nominators*, *era_depth*)

  Starts an Era and creates pending rewards for validator and nominators.

- Timer(*function*)

  Measures the time from the start of the specified f unction to its completion.

- Add-To(*collection*, *time*)

  Adds a returned time measurement (time) to collection.

- Compute-Weight(*collection*)

  Computes the resulting weight based on the time measurements in the collection. The worst case scenario should be chosen (the highest value).

8.5.3. Practical Example #3: `transfer`

The *`transfer`* function of the `balances` module is designed to move the specified balance by the sender to the receiver.

8.5.3.1. Analysis

The source code of this function is quite short:

```
let transactor = ensure_signed(origin)?;
let dest = T::Lookup::lookup(dest)?;
<Self as Currency<_>>::transfer(
  &transactor,
  &dest,
  value,
  ExistenceRequirement::AllowDeath
)?;
```

However, one need to pay close attention to the property `AllowDeath` and to how the function treat existingand non-existing accounts differently. Two types of behaviors are to consider:

- If the transfer completely depletes the sender account balance to zero (or bellow the minimum "keep-alive" requirement), it removes the address and all associated data from storage.
- If recipient account has no balance, the transfer also needs to create the recipient account.

8.5.3.2. Considerations

Specific parameters can could have a significant impact for this specific function. In order to trigger the two behaviors mentioned above, the following parameters are selected:

Type		From	To	Description
Account index	`index` in…	1	1000	Used as a seed for account creation
Balance	`balance` in…	2	1000	Sender balance and transfer amount

Executing a benchmark for each balance increment within the balance range for each index increment within the index range will generate too many variants (1000×999) and highly increase execution time. Therefore, this benchmark is configured to first set the balance at value 1'000 and then to iterate from 1 to 1'000 for the index value. Once the index value reaches 1'000, the balance value will reset to 2 and iterate to 1'000 (see Algorithm 27 for more detail):

- `index`: 1, `balance`: 1000
- `index`: 2, `balance`: 1000
- `index`: 3, `balance`: 1000

- ...
- `index`: 1000, `balance`: 1000
- `index`: 1000, `balance`: 2
- `index`: 1000, `balance`: 3
- `index`: 1000, `balance`: 4
- ...

The parameters itself do not influence or trigger the two worst conditions and must be handled by the implemented benchmarking tool. The $transfer$ benchmark is implemented as defined in Algorithm 27.

8.5.3.3. Benchmarking Framework

The benchmarking implementation for the Polkadot Runtime function $transfer$ is defined as follows (starting with the Main function):

Algorithm 27 "transfer" Runtime function benchmark

Ensure: collection: a collection of time measurements of all benchmark iterations

```
function Main()
    init collection = {}
    init balance = 1'000
    for index ← 1, 1'000 do
        time ← Run-Benchmark(index, balance)
        Add-To(collection, time)
    end for
    init index = 1'000
    for balance ← 2, 1'000 do
        time ← Run-Benchmark(index, balance)
        Add-To(collection, time)
    end for
    W ← Compute-Weight(collection)
    return W
end function
function Run-Benchmark(index, balance)
    sender ← Create-Account("caller", index)
    recipient ← Create-Account("recipient", index)
    Set-Balance(sender, balance)
    time ← Timer(Transfer(sender, recipient, balance))
    return time
end function
```

Algorithm 27. "transfer" Runtime function benchmark

where

- Create-Account($name$, $index$)

  Creates a Blake2 hash of the concatenated input of name and index representing the address of a account. This function only creates an address and does not conduct any I/O.

- Set-Balance($account$, $balance$)

  Sets a initial balance for the specified account in the storage state.

- Transfer($sender$, $recipient$, $balance$)

  Transfers the specified balance from sender to recipient by calling the corresponding Runtime function. This represents the target Runtime function to be benchmarked.

- Add-To($collection$, $time$)

  Adds a returned time measurement (time) to collection.

- Timer($function$)

  Adds a returned time measurement (time) to collection.

- Compute-Weight($collection$)

  Computes the resulting weight based on the time measurements in the collection. The worst case scenario should be chosen (the highest value).

8.5.4. Practical Example #4: `withdraw_unbounded`

The `withdraw_unbonded` function of the `staking` module is designed to move any unlocked funds from

the staking management system to be ready for transfer. It contains some operations which have some I/O overhead.

8.5.4.1. Analysis

Similarly to the `payout_stakers` function (Section 8.5.2), this function fetches the Ledger which contains information about the stash, such as bonded balance and unlocking balance (balance that will eventually be freed and can be withdrawn).

```
if let Some(current_era) = Self::current_era() {
  ledger = ledger.consolidate_unlocked(current_era)
}
```

The function `consolidate_unlocked` does some cleaning up on the ledger, where it removes outdated entries from the unlocking balance (which implies that balance is now free and is no longer awaiting unlock).

```
let mut total = self.total;
let unlocking = self.unlocking.into_iter()
  .filter(|chunk| if chunk.era > current_era {
    true
  } else {
    total = total.saturating_sub(chunk.value);
    false
  })
  .collect();
```

This function does a check on wether the updated ledger has any balance left in regards to staking, both in terms of locked, staking balance and unlocking balance. If not amount is left, the all information related to the stash will be deleted. This results in multiple I/O calls.

```
if ledger.unlocking.is_empty() && ledger.active.is_zero() {
  // This account must have called `unbond()` with some value that caused the active
  // portion to fall below existential deposit + will have no more unlocking chunks
  // left. We can now safely remove all staking-related information.
  Self::kill_stash(&stash, num_slashing_spans)?;
  // remove the lock.
  T::Currency::remove_lock(STAKING_ID, &stash);
  // This is worst case scenario, so we use the full weight and return None
  None
```

The resulting call to `Self::kill_stash()` triggers:

```
slashing::clear_stash_metadata::<T>(stash, num_slashing_spans)?;
<Bonded<T>>::remove(stash);
<Ledger<T>>::remove(&controller);
<Payee<T>>::remove(stash);
<Validators<T>>::remove(stash);
<Nominators<T>>::remove(stash);
```

Alternatively, if there's some balance left, the adjusted ledger simply gets updated back into storage.

```
// This was the consequence of a partial unbond. just update the ledger and move on.
Self::update_ledger(&controller, &ledger);
```

Finally, it withdraws the unlocked balance, making it ready for transfer:

```
let value = old_total - ledger.total;
Self::deposit_event(RawEvent::Withdrawn(stash, value));
```

8.5.4.2. Parameters

The following parameters are selected:

Type		From	To	Description
Account index	`index` in...	0	1000	Used as a seed for account creation

This benchmark does not require complex parameters. The values are used solely for account generation.

8.5.4.3. Considerations

Two important points in the `withdraw_unbonded` function must be considered. The benchmarks should trigger both conditions

- The updated ledger is inserted back into storage.
- If the stash gets killed, then multiple, repetitive deletion calls are performed in the storage.

8.5.4.4. Benchmarking Framework

The benchmarking implementation for the Polkadot Runtime function `withdraw_unbonded` is defined as follows:

Algorithm 28 "withdraw_unbonded" Runtime function benchmark

Ensure: W

```
function Main()
    init collection = {}
    for balance ← 1, 100 do
        stash ← Create-Account("stash", 1)
        controller ← Create-Account("controller", 1)
        Set-Balance(stash, 100)
        Set-Balance(controller, 1)
        Bond(stash, controller, balance)
        Pass-Era()
        UnBond(controller, balance)
        Pass-Era()
        time ← Timer(Withdraw-Unbonded(controller))
        Add-To(collection, time)
    end for
    W ← Compute-Weight(collection)
    return W
end function
```

Algorithm 28. "withdraw_unbonded" Runtime function benchmark

where

- Create-Account(*name*, *index*)

  Creates a Blake2 hash of the concatenated input of name and index representing the address of a account. This function only creates an address and does not conduct any I/O.

- Set-Balance(*account*, *balance*)

  Sets a initial balance for the specified account in the storage state.

- Bond(*stash*, *controller*, *amount*)

  Bonds the specified amount for the stash and controller pair.

- UnBond(*account*, *amount*)

  Unbonds the specified amount for the given account.

- Pass-Era()

  Pass one era. Forces the function `withdraw_unbonded` to update the ledger and eventually delete information.

- Withdraw-Unbonded(*controller*)

  Withdraws the the full unbonded amount of the specified controller account. This represents the target Runtime function to be benchmarked.

- Add-To(*collection*, *time*)

  Adds a returned time measurement (time) to collection.

- Timer(*function*)

  Measures the time from the start of the specified f unction to its completion.

- Compute-Weight(*collection*)

  Computes the resulting weight based on the time measurements in the collection. The worst case scenario should be chosen (the highest value).

8.6. Fees

Block producers charge a fee in order to be economically sustainable. That fee must always be covered by the sender of the transaction. Polkadot has a flexible mechanism to determine the minimum cost to include transactions in a block.

8.6.1. Fee Calculation

Polkadot fees consists of three parts:

- Base fee: a fixed fee that is applied to every transaction and set by the Runtime.
- Length fee: a fee that gets multiplied by the length of the transaction, in bytes.
- Weight fee: a fee for each, varying Runtime function. Runtime implementers need to implement a conversion mechanism which determines the corresponding currency amount for the calculated weight.

The final fee can be summarized as:

$$\begin{aligned} fee = & \, base\,fee \\ & + length\,of\,transaction\,in\,bytes \times length\,fee \\ & + weight\,to\,fee \end{aligned}$$

8.6.2. Definitions in Polkadot

The Polkadot Runtime defines the following values:

- Base fee: 100 uDOTs
- Length fee: 0.1 uDOTs
- Weight to fee conversion:

$$weight\,fee = weight \times (100 uDOTs \div (10 \times 10\,'\,000))$$

  A weight of 10’000 (the smallest non-zero weight) is mapped to $\frac{1}{10}$ of 100 uDOT. This fee will never exceed the max size of an unsigned 128 bit integer.

8.6.3. Fee Multiplier

Polkadot can add a additional fee to transactions if the network becomes too busy and starts to decelerate the system. This fee can create an incentive to avoid the production of low priority or insignificant transactions. In contrast, those additional fees will decrease if the network calms down and it can execute transactions without much difficulties.

That additional fee is known as the `Fee Multiplier` and its value is defined by the Polkadot Runtime. The multiplier works by comparing the saturation of blocks; if the previous block is less saturated than the current block (implying an uptrend), the fee is slightly increased. Similarly, if the previous block is more saturated than the current block (implying a downtrend), the fee is slightly decreased.

The final fee is calculated as:

$$final\,fee = fee \times Fee\,Multiplier$$

8.6.3.1. Update Multiplier

The `Update Multiplier` defines how the multiplier can change. The Polkadot Runtime internally updates the multiplier after each block according the following formula:

$$\begin{aligned} diff &= (target\,weight - previous\,block\,weight) \\ v &= 0.00004 \\ next\,weight &= weight \times (1 + (v \times diff) + (v \times diff)^2 / 2) \end{aligned}$$

Polkadot defines the `target_weight` as 0.25 (25%). More information about this algorithm is described in the Web3 Foundation research paper.

Chapter 9. Consensus

9.1. BABE digest messages

The Runtime is required to provide the BABE authority list and randomness to the host via a consensus message in the header of the first block of each epoch.

The digest published in Epoch $\mathcal{E}_n$ is enacted in $\mathcal{E}_{n+1}$. The randomness in this digest is computed based on the all the VRF outputs up to including Epoch $\mathcal{E}_{n-2}$ while the authority set is based on all transaction included up to Epoch $\mathcal{E}_{n-1}$.

The computation of the randomeness seed is described in Algorithm 29 which uses the concept of epoch subchain as described in host specification and the value d_B, which is the VRF output computed for slot s_B.

Algorithm 29 Epoch-Randomness

Require: n > 2

1: **init** ρ ← ϕ
2: **for** B in SubChain(E_{n-2}) **do**
3: ρ ← ρ||d_B
4: **end for**
5: **return** Blake2b(Epoch-Randomness(s_{n-1})||n||ρ)

Algorithm 29. Epoch-Randomness

where tem:[n] is the epoch index.

Additional information

Appendix chapter containing various protocol details

Chapter 10. Cryptographic Algorithms

10.1. Hash Functions

10.2. 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 the SHA-3 competition.

Polkadot is using the Blake2b variant which is optimized for 64-bit platforms. Unless otherwise specified, the Blake2b hash function with a 256-bit 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 [1].

10.3. Randomness

TBH

10.4. VRF

A Verifiable Random Function (VRF) is a mathematical operation that takes some input and produces a random number using a secret key along with a proof of authenticity that this random number was generated using the submitter's secret key and the given input. The proof can be verified by any challenger to ensure the random number generation is valid and has not been tampered with (for example to the benfit of submitter).

In Polkadot, VRFs are used for the BABE block production lottery by Algorithm 6 and the parachain approval voting mechanism (Section 6.5). The VRF uses mechanism similar to algorithms introduced in the following papers:

- Making NSEC5 Practical for DNSSEC.
- DLEQ Proofs.
- Verifiable Random Functions (VRFs).

It essentially generates a deterministic elliptic curve based Schnorr signature as a verifiable random value. The elliptic curve group used in the VRF function is the Ristretto group specified in:

- https://ristretto.group/

Definition 147. VRF Proof

The **VRF proof** proves the correctness for an associated VRF output. The VRF proof, P, is a datastructure of the following format:

$$P = (C, S) S = (b_0, \text{ellipsis}\, b_{31})$$

where C is the challenge and S is the 32-byte Schnorr poof. Both are expressed as Curve25519 scalars as defined in Definition Definition 148.

Definition 148. DLEQ Prove

The $\text{dleq\_prove}(t, i)$ function creates a proof for a given input, i, based on the provided transcript, T.

First:

$$t_1 = \text{append}(t, \text{'proto-name'}, \text{'DLEQProof'}) t_2 = \text{append}(t_1, \text{'vrf:h'}, i)$$

Then the witness scalar is calculated, s_w, where w is the 32-byte secret seed used for nonce generation in the context of sr25519.

$$t_3 = \text{meta-AD}(t_2, \text{'proving\00'}, \text{more=False}) t_4 = \text{meta-AD}(t_3, w_l, \text{more=True}) t_5 = \text{KEY}(t_4, w, \text{more=False}) t_6 = \text{meta-AD}(t_5, \text{'rng'}, \text{more=False}) t_7 = \text{KEY}(t_6, r, \text{more=False}) t_8 = \text{meta-AD}(t_7, e_{64}, \text{more=False}) (\phi, s_w) = \text{PRF}(t_8, \text{more=False})$$

where w_l is length of the witness, encoded as a 32-bit little-endian integer. r is a 32-byte array containing the secret witness scalar.

$$l_1 = \text{append}(t_2, \text{'vrf:R=g\^{}r'}, s_w) l_2 = \text{append}(l_1, \text{'vrf:h\^{}r'}, s_i) l_3 = \text{append}\big(l_2, \text{'vrf:pk'}, s_p\big) l_4 = \text{append}(l_3, \text{'vrf:h\^{}sk'}, \text{vrf}_o)$$

where

- s_i is the compressed Ristretto point of the scalar input.
- s_p is the compressed Ristretto point of the public key.
- s_w is the compressed Ristretto point of the wittness:

For the 64-byte challenge:

$$l_5 = \text{meta-AD}(l_4, \text{'prove'}, \text{more=False}) l_6 = \text{meta-AD}(l_5, e_{64}, \text{more=True}) C = \text{PRF}(l_6, \text{more=False})$$

And the Schnorr proof:

$$S = s_w - (C \cdot p)$$

where p is the secret key.

Definition 149. DLEQ Verify

The dleq_verify(i, o, P, p_k) function verifiers the VRF input, i against the output, o, with the associated proof (Definition 147) and public key, p_k.

$t_1 = \text{append}(t, \text{'proto-name'}, \text{'DLEQProof'}) t_2 = \text{append}(t_1, \text{'vrf:h'}, s_l) t_3 = \text{append}(t_2, \text{'vrf:R=g^r'}, R) t_4 = \text{append}(t_3, \text{'vrf:h^r'}, H) t_5 = \text{append}(t_4, \text{'vrf:pk'}, p_k) t_6 = \text{append}(t_5, \text{'vrf:h^sk'}, o)$

where

- R is calculated as:

$$R = C \in P \times p_k + S \in P + B$$

  where B is the Ristretto basepoint.

- H is calculated as:

$$H = C \in P \times o + S \in P \times i$$

The challenge is valid if $C \in P$ equals y:

$$t_7 = \text{meta-AD}(t_6, \text{'prove'}, \text{more=False}) t_8 = \text{meta-AD}(t_7, e_{64}, \text{more=True}) \, y = \text{PRF}(t_8, \text{more=False})$$

10.4.1. Transcript

A VRF transcript serves as a domain-specific separator of cryptographic protocols and is represented as a mathematical object, as defined by Merlin, which defines how that object is generated and encoded. The usage of the transcript is implementation specific, such as for certain mechanisms in the Availability & Validity chapter (Chapter 6), and is therefore described in more detail in those protocols. The input value used to generate the transactions is referred to as a *context* ([defn-vrf-context]).

Definition 150. VRF Transcript

A **transcript**, or VRF transcript, is a STROBE object, obj, as defined in the STROBE documentation, respectively section "5. State of a STROBE object".

$$\text{obj} = (\text{st}, \text{pos}, \text{pos}_{\text{begin}}, I_0)$$

where:

- The duplex state, st, is a 200-byte array created by the keccak-f1600 sponge function on the initial STROBE state. Specifically, `R` is of value `166` and `X.Y.Z` is of value `1.0.2`.
- pos has the initial value of `0`.
- $\text{pos}_{\text{begin}}$ has the initial value of `0`.
- I_0 has the initial value of `0`.

Then, the `meta-AD` operation (Definition 151) (where `more=False`) is used to add the protocol label `Merlin v1.0` to obj followed by *appending* (Section 10.4.1.1) label `dom-step` and its corresponding context, ctx, resulting in the final transcript, T.

$$t = \text{meta-AD}(obj, \text{'Merlin v1.0'}, \text{False}) T = \text{append}(t, \text{'dom-step'}, \text{ctx})$$

ctx serves as an arbitrary identifier/separator and its value is defined by the protocol specification individually. This transcript is treated just like a STROBE object, wherein any operations (Definition 151) on it modify the values such as pos and $\text{pos}_{\text{begin}}$.

Formally, when creating a transcript we refer to it as $\text{Transcript}(ctx)$.

Definition 151. STROBE Operations

STROBE operations are described in the STROBE specification, respectively section "6. Strobe operations". Operations are indicated by their corresponding bitfield, as described in section "6.2. Operations and flags" and implemented as described in section "7. Implementation of operations"

10.4.1.1. Appending Messages

Appending messages, or "data", to the transcript (Definition 150) first requires `meta-AD` operations for a given label of the messages, including the size of the message, followed by an `AD` operation on the message itself. The size of the message is a 4-byte, little-endian encoded integer.

$$T_0 = \text{meta-AD}(T, l, \text{False}) T_1 = \text{meta-AD}(T_0, m_l, \text{True}) T_2 = \text{AD}(T_1, m, \text{False})$$

where T is the transcript (Definition 150), l is the given label and m the message, respectively m_l representing its size. T_2 is the resulting transcript with the appended data. STROBE operations are described in Definition 151.

Formally, when appending a message we refer to it as $\text{append}(T, l, m)$.

10.5. Cryptographic Keys

Various types of keys are used in Polkadot to prove the identity of the actors involved in the 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 152. Account Key

Account key (sk^a, pk^a) is a key pair of type of either of the schemes in the following table:

Table 2. List of the public key scheme which can be used for an account key

Key Scheme	Description
sr25519	Schnorr signature on Ristretto compressed ed25519 points as implemented in TODO
ed25519	The standard ed25519 signature complying with TODO
secp256k1	Only for outgoing transfer transactions.

An account key can be used to sign transactions among other accounts and balance-related functions. There are two prominent subcategories of account keys namely "stash keys" and "controller keys", each being used for a different function. Keys defined in Definitions Definition 152, Definition 153 and Definition 154 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 link_sect-creating-controller-key[9.5.2] and link_sect-certifying-keys[9.5.5] respectively.

Definition 153. Stash Key

The **Stash key** is a type of account key that holds funds bonded for staking (described in Section link_sect-staking-funds[9.5.1]) to a particular controller key (defined in Definition Definition 154). 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 link_sect-creating-controller-key[9.5.2]. 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 154. Controller Key

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 the other keys. It is a key that will be in 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 link_sect-controller-settings[9.5.4]. If used for a validator, it certifies the session keys, as described in link_sect-certifying-keys[9.5.5]. It only needs the required funds to pay transaction fees [TODO: key needing fund needs to be defined].

Definition 155. Session Keys

Session keys are short-lived keys that are used to authenticate validator operations. Session keys are generated by the Polkadot Host and should be changed regularly due to security reasons. Nonetheless, no validity period is enforced by the Polkadot protocol on session keys. Various types of keys used by the Polkadot Host are presented in Table link_tabl-session-keys[9.1]*:*

Table 3. List of key schemes which are used for session keys depending on the protocol

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

Session keys must be accessible by certain Polkadot Host APIs defined in Appendix link_sect-host-api[12]. Session keys are *not* meant to control the majority of the users' funds and should only be used for their intended purpose.

10.5.1. Holding and staking funds

TBH

10.5.2. Creating a Controller key

TBH

10.5.3. Designating a proxy for voting

TBH

10.5.4. Controller settings

TBH

10.5.5. Certifying keys

Due to security considerations and Runtime upgrades, the session keys are supposed to be changed regularly. As such, the new session keys need to be certified by a controller key before putting them in use. The controller only needs to create a certificate by signing a session public key and broadcasting this certificate via an extrinsic. [TODO: spec the detail of the data structure of the certificate etc.]

Chapter 11. Auxiliary Encodings

11.1. Binary Enconding

Definition 156. *Sequence of Bytes*

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

$$b := (b_0, b_1, \text{ellipsis}, b_{n-1}) \text{such that} 0 \& <; = b_i \& <; = 255$$

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

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

We represent the concatenation of byte arrays $a := (a_0, \text{ellipsis}, a_n)$ and $b := (b_0, \text{ellipsis}, b_m)$ by:

$$a||b := (a_0, \text{ellipsis}, a_n, b_0, \text{ellipsis}, b_m)$$

Definition 157. *Bitwise Representation*

For a given byte $0 \leq b \leq 255$ the **bitwise representation** in bits $b_i \in \{0, 1\}$ is defined as:

$$b := b_7 \text{ellipsis} b_0$$

where

$$b = 2^7 b_7 + 2^6 b_6 + \text{ellipsis} + 2^0 b_0$$

Definition 158. *Little Endian*

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

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

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

$$b_i := B_i$$

Accordingly, we define the function $\mathsf{Enc}_{\mathsf{LE}}$:

$$\mathsf{Enc}_{\mathsf{LE}} : \mathbb{Z}^+ - \& >; \mathbb{B}; (B_n \text{ellipsis} B_0)_{256} | - \& >; (B_0, B_1, \text{ellipsis}, B_n)$$

Definition 159. *UINT32*

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

11.2. SCALE Codec

The Polkadot Host 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 160. Decoding

$\text{Dec}_{\text{SC}}(d)$ refers to the decoding of a blob of data. Since the SCALE codec is not self-describing, it's up to the decoder to validate whether the blob of data can be deserialized into the given type or data structure.

It's accepted behavior for the decoder to partially decode the blob of data. Meaning, any additional data that does not fit into a datastructure can be ignored.

Considering that the decoded data is never larger than the encoded message, this information can serve as a way to validate values that can vary in sizes, such as sequences (Definition 166). The decoder should strictly use the size of the encoded data as an upper bound when decoding in order to prevent denial of service attacks.

Definition 161. Tuple

The **SCALE codec** for **Tuple**, T, such that:

$$T := (A_1, \text{ellipsis}\, A_n)$$

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

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

In case of a tuple (or a structure), 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 happening.

Definition 162. Varying Data Type

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

$$\mathcal{T} = \{T_1, \text{ellipsis}\, , T_n\}$$

A value A of varying date type is a pair $(A_{\text{Type}}, A_{\text{Value}})$ where $A_{\text{Type}} = T_i$ for some $T_i \in \mathcal{T}$ and A_{Value} is its value of type T_i, which can be empty. We define $\text{idx}(T_i) = i-1$, unless it is explicitly defined as another value in the definition of a particular varying data type.

In particular, we define two specific varying data which are frequently used in various part of Polkadot protocol: *Option* (Definition 164) and *Result* (Definition 165).

Definition 163. *Encoding of Varying Data Type*

The SCALE codec for value $A = (A_{\text{Type}}, A_{\text{Value}})$ of varying data type $\mathcal{T} = \{T_i, \text{ellipsis}\, T_n\}$, formally referred to as $\text{Enc}_{\text{SC}}(A)$ is defined as follows:

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

Where idx is encoded in a fixed length integer determining the type of A. In particular, for the optional type defined in Definition 162, we have:

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

The SCALE codec does not encode the correspondence between the value and the data type it represents; the decoder needs prior knowledge of such correspondence to decode the data.

Definition 164. *Option Type*

The **Option** type is a varying data type of $\{\text{None}, T_2\}$ which indicates if data of T_2 type is available (referred to as *some* state) or not (referred to as *empty*, *none* or *null* state). The presence of type *none*, indicated by $\text{idx}(T_{\text{None}}) = 0$, implies that the data corresponding to T_2 type is not available and contains no additional data. Where as the presence of type T_2 indicated by $\text{idx}(T_2) = 1$ implies that the data is available.

Definition 165. *Result Type*

The **Result** type is a varying data type of $\{T_1, T_2\}$ which is used to indicate if a certain operation or function was executed successfully (referred to as "ok" state) or not (referred to as "error" state). T_1 implies success, T_2 implies failure. Both types can either contain additional data or are defined as empty type otherwise.

Definition 166. *Sequence*

The **SCALE codec** for **sequence** S such that:

$$S : = A_1, \text{ellipsis}\, A_n$$

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:

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

where $\text{Enc}_{\text{SC}}^{\text{Len}}$ is defined in Definition 171.

In some cases, the length indicator $\text{Enc}_{\text{SC}}^{\text{Len}}(\text{abs}\, S)$ is omitted if the length of the sequence is fixed and known by the decoder upfront. Such cases are explicitly stated by the definition of the corresponding type.

Definition 167. *Dictionary*

SCALE codec for **dictionary** or **hashtable** D with key-value pairs (k_i, v_i)s such that:

$$D := \{(k_1, v_1), \text{ellipsis}\,(k_n, v_n)\}$$

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

$$\text{Enc}_{\text{SC}}(D) := \text{Enc}_{\text{SC}}^{\text{Size}}(\text{abs}\,D)||\text{Enc}_{\text{SC}}(k_1, v_1)||\text{ellipsis}\,||\text{Enc}_{\text{SC}}(k_n, v_n)$$

where $\text{Enc}_{\text{SC}}^{\text{Size}}$ is encoded the same way as $\text{Enc}_{\text{SC}}^{\text{Len}}$ but argument Size refers to the number of key-value pairs rather than the length.

Definition 168. *Boolean*

The **SCALE codec** for **boolean value** b defined as a byte as follows:

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

Definition 169. *Fixed Length*

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 158.

Definition 170. *Empty*

The SCALE codec, Enc_{SC}, for an empty type is defined to a byte array of zero length and depicted as ϕ.

11.2.1. Length and Compact Encoding

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

Definition 171. *Length Encoding*

SCALE Length encoding, $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$, also known as a *compact encoding*, of a non-negative number n is defined as follows:

$$\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}} : \mathbb{N} - \& > ; \mathbb{B}\, n - \& > ; b : = \begin{cases} l_1 & 0 \& < ; = n \& < ; 2^6 \\ i_1 i_2 & 2^6 \& < ; = n \& < ; 2^{14} \\ j_1 j_2 j_3 & 2^{14} \& < ; = n \& < ; 2^{30} \\ k_1 k_2 \mathrm{ellipsis}\, k_m & 2^{30} \& < ; = n \end{cases}$$

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

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

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

$$n : = \begin{cases} l_1^7 \mathrm{ellipsis}\, l_1^3 l_1^2 & n \& < ; 2^6 \\ i_2^7 \mathrm{ellipsis}\, i_2^0 i_1^7 .. i_1^2 & 2^6 \& < ; = n \& < ; 2^{14} \\ j_4^7 \mathrm{ellipsis}\, j_4^0 j_3^7 \mathrm{ellipsis}\, j_1^7 \mathrm{ellipsis}\, j_1^2 & 2^{14} \& < ; = n \& < ; 2^{30} \\ k_2 + k_3 2^8 + k_4 2^{2 \times 8} + \mathrm{ellipsis} + k_m 2^{(m-2)8} & 2^{30} \& < ; = n \end{cases}$$

such that:

$$k_1^7 \mathrm{ellipsis}\, k_1^3 k_1^2 : = m - 4$$

11.3. 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. To this aim, we define hex encoding function $\mathrm{Enc}(\mathrm{HE})(\mathrm{PK})$ as follows:

Definition 172. *Hex Encoding*

Suppose that $\mathrm{PK} = (k_1, \mathrm{ellipsis}\, k_n)$ is a sequence of nibbles, then:

$$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) : = \begin{cases} \mathrm{Nibbles}_4 & - \& > ; \quad \mathbb{B} \\ \mathrm{PK} = (k_1, \mathrm{ellipsis}\, k_n) & - \& > ; \quad \begin{cases} (16k_1 + k_2, \mathrm{ellipsis}\, , 16k_{2i-1} + k_{2i}) & n = 2i \\ (k_1, 16k_2 + k_3, \mathrm{ellipsis}\, , 16k_{2i} + k_{2i+1}) & n = 2i + 1 \end{cases} \end{cases}$$

Chapter 12. Genesis State

The genesis state is a set of key-value pairs representing the initial state of the Polkadot state storage. It can be retrieved from the Polkadot repository. While each of those key-value pairs offers important identifiable information to the Runtime, to the Polkadot Host they are a transparent set of arbitrary chain- and network-dependent keys and values. The only exception to this are the `:code` (Section 3.1.1) and `:heappages` (Section 3.1.2.1) keys, which are used by the Polkadot Host to initialize the WASM environment and its Runtime. The other keys and values are unspecified and solely depend on the chain and respectively its corresponding Runtime. On initialization the data should be inserted into the state storage with the Host API (Section 15.1.1).

As such, Polkadot does not defined a formal genesis block. Nonetheless for the compatibility reasons in several algorithms, the Polkadot Host defines the *genesis header* ([defn-genesis-header]). By the abuse of terminology, "genesis block" refers to the hypothetical parent of block number *1* which holds genisis header as its header.

The Polkadot genesis header is a data structure conforming to block header format (Definition 28). It contains the following values:

Block header field	**Genesis Header Value**
`parent_hash`	*0*
`number`	*0*
`state_root`	Merkle hash of the state storage trie (Definition 24) after inserting the genesis state in it.
`extrinsics_root`	*0*
`digest`	*0*

Chapter 13. Erasure Encoding

13.1. Erasure Encoding

Erasure Encoding has not been documented yet.

Host API

Description of the expected environment available for import by the Polkadot Runtime

Definition 173. *State Version*

The state version, ν, dictates how a merkle root should be constructed. The datastructure is a varying type of the following format:

$$\nu = \begin{cases} 0 & \text{full values} \\ 1 & \text{node hashes} \end{cases}$$

where 0 indicates that the values of the keys should be inserted into the trie directly and 1 makes use of "node hashes" when calculating the merkle proof (Definition 23).

Chapter 14. Preliminaries

The Polkadot Host API is a set of functions that the Polkadot Host 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. The encoding of each data type is specified or referenced in this section. If the encoding is not mentioned, then the default Wasm encoding is used, such as little-endian byte ordering for integers.

Definition 174. *Exposed Host API*

By RE_B we refer to the API exposed by the Polkadot Host which interact, manipulate and response based on the state storage whose state is set at the end of the execution of block B.

Definition 175. *Runtime Pointer*

The **Runtime pointer** type is an unsigned 32-bit integer representing a pointer to data in memory. This pointer is the primary way to exchange data of fixed/known size between the Runtime and Polkadot Host.

Definition 176. *Runtime Pointer Size*

The **Runtime pointer-size** type is an unsigned 64-bit integer, representing two consecutive integers. The least significant is **Runtime pointer** (Definition 175). The most significant provides the size of the data in bytes. This representation is the primary way to exchange data of arbitrary/dynamic sizes between the Runtime and the Polkadot Host.

Definition 177. *Lexicographic ordering*

Lexicographic ordering refers to the ascending ordering of bytes or byte arrays, such as:

$$[0, 0, 2] \& < ; [0, 1, 1] \& < ; [1] \& < ; [1, 1, 0] \& < ; [2] \& < ; [\mathrm{ellipsis}]$$

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

Chapter 15. Storage

Interface for accessing the storage from within the runtime.

15.1. Functions

15.1.1. `ext_storage_set`

Sets the value under a given key into storage.

15.1.1.1. Version 1 - Prototype

```
(func $ext_storage_set_version_1
    (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.
- `value`: a pointer-size (Definition 176) containing the value.

15.1.2. `ext_storage_get`

Retrieves the value associated with the given key from storage.

15.1.2.1. Version 1 - Prototype

```
(func $ext_storage_get_version_1
    (param $key i64) (result i64))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.
- `result`: a pointer-size (Definition 176) returning the SCALE encoded *Option* value (Definition 164) containing the value.

15.1.3. `ext_storage_read`

Gets the given key from storage, placing the value into a buffer and returning the number of bytes that the entry in storage has beyond the offset.

15.1.3.1. Version 1 - Prototype

```
(func $ext_storage_read_version_1
    (param $key i64) (param $value_out i64) (param $offset u32) (result i64))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.
- `value_out`: a pointer-size (Definition 176) containing the buffer to which the value will be written to. This function will never write more then the length of the buffer, even if the value's length is bigger.
- `offset`: an u32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition 176) pointing to a SCALE encoded *Option* value (Definition 164) containing an unsigned 32-bit integer representing the number of bytes left at supplied `offset`. Returns *None* if the entry does not exists.

15.1.4. `ext_storage_clear`

Clears the storage of the given key and its value. Nonexistent entries are silently ignored.

15.1.4.1. Version 1 - Prototype

```
(func $ext_storage_clear_version_1
    (param $key_data i64))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.

15.1.5. `ext_storage_exists`

Checks whether the given key exists in storage.

15.1.5.1. Version 1 - Prototype

```
(func $ext_storage_exists_version_1
    (param $key_data i64) (return i32))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.
- `return`: an i32 integer value equal to *1* if the key exists or a value equal to *0* if otherwise.

15.1.6. `ext_storage_clear_prefix`

Clear the storage of each key/value pair where the key starts with the given prefix.

15.1.6.1. Version 1 - Prototype

```
(func $ext_storage_clear_prefix_version_1
    (param $prefix i64))
```

Arguments

- `prefix`: a pointer-size (Definition 176) containing the prefix.

15.1.6.2. Version 2 - Prototype

```
(func $ext_storage_clear_prefix_version_2
    (param $prefix i64) (param $limit i64)
    (return i64))
```

Arguments

- `prefix`: a pointer-size (Definition 176) containing the prefix.
- `limit`: a pointer-size (Definition 176) to an *Option* type (Definition 164) containing an unsigned 32-bit integer indicating the limit on how many keys should be deleted. No limit is applied if this is *None*. Any keys created during the current block execution do not count towards the limit.
- `return`: a pointer-size (Definition 176) to the following variant, k:

$$k = \begin{cases} 0 & -\&>;c \\ 1 & -\&>;c \end{cases}$$

  where *0* indicates that all keys of the child storage have been removed, followed by the number of removed keys, c. The variant *1* indicates that there are remaining keys, followed by the number of removed keys.

15.1.7. `ext_storage_append`

Append the SCALE encoded value to a SCALE encoded sequence (Definition 166) at the given key. This function assumes that the existing storage item is either empty or a SCALE encoded sequence and that the value to append is also SCALE encoded and of the same type as the items in the existing sequence.

To improve performance, this function is allowed to skip decoding the entire SCALE encoded sequence and instead can just append the new item to the end of the existing data and increment the length prefix $\text{Enc}_{\text{SC}}^{\text{Len}}$.

If the storage item does not exist or is not SCALE encoded, the storage item will be set to the specified value, represented as a SCALE encoded byte array.

15.1.7.1. Version 1 - Prototype

```
(func $ext_storage_append_version_1
    (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size (Definition 176) containing the key.

- `value`: a pointer-size (Definition 176) containing the value to be appended.

15.1.8. `ext_storage_root`

Compute the storage root.

15.1.8.1. Version 1 - Prototype

```
(func $ext_storage_root_version_1
    (return i64))
```

Arguments

- `return`: a pointer-size (Definition 176) to a buffer containing the 256-bit Blake2 storage root.

15.1.8.2. Version 2 - Prototype

```
(func $ext_storage_root_version_2
    (param $version i32) (return i32))
```

Arguments

- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 storage root.

15.1.9. `ext_storage_changes_root`

This function is not longer used and only exists for compatibility reasons.

15.1.9.1. Version 1 - Prototype

```
(func $ext_storage_changes_root_version_1
    (param $parent_hash i64) (return i64))
```

Arguments

- `parent_hash`: a pointer-size (Definition 176) to the SCALE encoded block hash.
- `return`: a pointer-size (Definition 176) to an *Option* type (Definition 164) that's always *None*.

15.1.10. `ext_storage_next_key`

Get the next key in storage after the given one in lexicographic order (Definition 177). The key provided to this function may or may not exist in storage.

15.1.10.1. Version 1 - Prototype

```
(func $ext_storage_next_key_version_1
    (param $key i64) (return i64))
```

Arguments

- `key`: a pointer-size (Definition 176) to the key.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the next key in lexicographic order.

15.1.11. `ext_storage_start_transaction`

Start a new nested transaction. This allows to either commit or roll back all changes that are made after this call. For every transaction there must be a matching call to either `ext_storage_rollback_transaction` (Section 15.1.12) or `ext_storage_commit_transaction` (Section 15.1.13). This is also effective for all values manipulated using the child storage API (Chapter 16).

This is a low level API that is potentially dangerous as it can easily result in unbalanced transactions. Runtimes should use high level storage abstractions.

15.1.11.1. Version 1 - Prototype

```
(func $ext_storage_start_transaction_version_1)
```

Arguments

- None.

15.1.12. `ext_storage_rollback_transaction`

Rollback the last transaction started by `ext_storage_start_transaction` (Section 15.1.11). Any changes made during that transaction are discarded.

Panics if `ext_storage_start_transaction` (Section 15.1.11) was not called.

15.1.12.1. Version 1 - Prototype

```
(func $ext_storage_rollback_transaction_version_1)
```

Arguments

- None.

15.1.13. `ext_storage_commit_transaction`

Commit the last transaction started by `ext_storage_start_transaction` (Section 15.1.11). Any changes made during that transaction are committed to the main state.

Panics if `ext_storage_start_transaction` (Section 15.1.11) was not called.

15.1.13.1. Version 1 - Prototype

```
(func $ext_storage_commit_transaction_version_1)
```

Arguments

- None.

Chapter 16. Child Storage

Interface for accessing the child storage from within the runtime.

Definition 178. Child Storage

Child storage key is a unprefixed location of the child trie in the main trie.

16.1. Functions

16.1.1. `ext_default_child_storage_set`

Sets the value under a given key into the child storage.

16.1.1.1. Version 1 - Prototype

```
(func $ext_default_child_storage_set_version_1
    (param $child_storage_key i64) (param $key i64) (param $value i64))
```

Arguments

- `child_storage_key` : a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.

16.1.2. `ext_default_child_storage_get`

Retrieves the value associated with the given key from the child storage.

16.1.2.1. Version 1 - Prototype

```
(func $ext_default_child_storage_get_version_1
    (param $child_storage_key i64) (param $key i64) (result i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the value.

16.1.3. `ext_default_child_storage_read`

Gets the given key from storage, placing the value into a buffer and returning the number of bytes that the entry in storage has beyond the offset.

16.1.3.1. Version 1 - Prototype

```
(func $ext_default_child_storage_read_version_1
    (param $child_storage_key i64) (param $key i64) (param $value_out i64)
    (param $offset u32) (result i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.
- `value_out`: a pointer-size (Definition 176) to the buffer to which the value will be written to. This function will never write more then the length of the buffer, even if the value's length is bigger.
- `offset`: an u32 integer containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the number of bytes written into the **value_out** buffer. Returns if the entry does not exists.

16.1.4. `ext_default_child_storage_clear`

Clears the storage of the given key and its value from the child storage. Nonexistent entries are silently ignored.

16.1.4.1. Version 1 - Prototype

```
(func $ext_default_child_storage_clear_version_1
    (param $child_storage_key i64) (param $key i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.

16.1.5. `ext_default_child_storage_storage_kill`

Clears an entire child storage.

16.1.5.1. Version 1 - Prototype

```
(func $ext_default_child_storage_storage_kill_version_1
    (param $child_storage_key i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).

16.1.5.2. Version 2 - Prototype

```
(func $ext_default_child_storage_storage_kill_version_2
    (param $child_storage_key i64) (param $limit i64)
    (return i32))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `limit`: a pointer-size (Definition 176) to an *Option* type (Definition 164) containing an unsigned 32-bit integer indicating the limit on how many keys should be deleted. No limit is applied if this is *None*. Any keys created during the current block execution do not count towards the limit.
- `return`: a value equal to *1* if all the keys of the child storage have been deleted or a value equal to *0* if there are remaining keys.

16.1.5.3. Version 3 - Prototype

```
(func $ext_default_child_storage_storage_kill_version_3
    (param $child_storage_key i64) (param $limit i64)
    (return i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `limit`: a pointer-size (Definition 176) to an *Option* type (Definition 164) containing an unsigned 32-bit integer indicating the limit on how many keys should be deleted. No limit is applied if this is *None*. Any keys created during the current block execution do not count towards the limit.
- `return`: a pointer-size (Definition 176) to the following variant, k:

$$k = \begin{cases} 0 & -\&>;c \\ 1 & -\&>;c \end{cases}$$

  where *0* indicates that all keys of the child storage have been removed, followed by the number of removed keys, c. The variant *1* indicates that there are remaining keys, followed by the number of removed keys.

16.1.6. `ext_default_child_storage_exists`

Checks whether the given key exists in the child storage.

16.1.6.1. Version 1 - Prototype

```
(func $ext_default_child_storage_exists_version_1
    (param $child_storage_key i64) (param $key i64) (return i32))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.
- `return`: an i32 integer value equal to *1* if the key exists or a value equal to *0* if otherwise.

16.1.7. `ext_default_child_storage_clear_prefix`

Clears the child storage of each key/value pair where the key starts with the given prefix.

16.1.7.1. Version 1 - Prototype

```
(func $ext_default_child_storage_clear_prefix_version_1
    (param $child_storage_key i64) (param $prefix i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `prefix`: a pointer-size (Definition 176) to the prefix.

16.1.7.2. Version 2 - Prototype

```
(func $ext_default_child_storage_clear_prefix_version_2
    (param $child_storage_key i64) (param $prefix i64)
    (param $limit i64) (return i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `prefix`: a pointer-size (Definition 176) to the prefix.
- `limit`: a pointer-size (Definition 176) to an *Option* type (Definition 164) containing an unsigned 32-bit integer indicating the limit on how many keys should be deleted. No limit is applied if this is *None*. Any keys created during the current block execution do not count towards the limit.
- `return`: a pointer-size (Definition 176) to the following variant, k:

$$k = \begin{cases} 0 & -\&>; c \\ 1 & -\&>; c \end{cases}$$

  where *0* indicates that all keys of the child storage have been removed, followed by the number of removed keys, c. The variant *1* indicates that there are remaining keys, followed by the number of removed keys.

16.1.8. `ext_default_child_storage_root`

Commits all existing operations and computes the resulting child storage root.

16.1.8.1. Version 1 - Prototype

```
(func $ext_default_child_storage_root_version_1
    (param $child_storage_key i64) (return i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `return`: a pointer-size (Definition 176) to the SCALE encoded storage root.

16.1.8.2. Version 2 - Prototype

```
(func $ext_default_child_storage_root_version_2
    (param $child_storage_key i64) (param $version i32)
    (return i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `version`: a pointer-size (Definition 175) to the state version (Definition 173).
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 storage root.

16.1.9. `ext_default_child_storage_next_key`

Gets the next key in storage after the given one in lexicographic order (Definition 177). The key provided to this function may or may not exist in storage.

16.1.9.1. Version 1 - Prototype

```
(func $ext_default_child_storage_next_key_version_1
    (param $child_storage_key i64) (param $key i64) (return i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 176) to the child storage key (Definition 178).
- `key`: a pointer-size (Definition 176) to the key.
- `return`: a pointer-size (Definition 176) to the SCALE encoded as defined in Definition 164 containing the next key in lexicographic order. Returns if the entry cannot be found.

Chapter 17. Crypto

Interfaces for working with crypto related types from within the runtime.

Definition 179. Key Type Identifier

Cryptographic keys are stored in separate key stores based on their intended use case. The separate key stores are identified by a 4-byte ASCII **key type identifier**. The following known types are available:

Table 4. Table of known key type identifiers

Id	Description
acco	Key type for the controlling accounts
babe	Key type for the Babe module
gran	Key type for the Grandpa module
imon	Key type for the ImOnline module
audi	Key type for the AuthorityDiscovery module
para	Key type for the Parachain Validator Key
asgn	Key type for the Parachain Assignment Key

Definition 180. ECDSA Verify Error

EcdsaVerifyError is a varying data type (Definition 162) that specifies the error type when using ECDSA recovery functionality. Following values are possible:

Table 5. Table of error types in ECDSA recovery

Id	Description
0	Incorrect value of R or S
1	Incorrect value of V
2	Invalid signature

17.1. Functions

17.1.1. `ext_crypto_ed25519_public_keys`

Returns all *ed25519* public keys for the given key identifier from the keystore.

17.1.1.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_public_keys_version_1
    (param $key_type_id i32) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key type identifier (Definition 179).
- `return`: a pointer-size (Definition 176) to an SCALE encoded 256-bit public keys.

17.1.2. `ext_crypto_ed25519_generate`

Generates an *ed25519* key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

17.1.2.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_generate_version_1
    (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key type identifier (Definition 179).
- `seed`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the BIP-39 seed which must be valid UTF8.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit public key.

17.1.3. `ext_crypto_ed25519_sign`

Signs the given message with the `ed25519` key that corresponds to the given public key and key type in the keystore.

17.1.3.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_sign_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key type identifier (Definition 179).
- `key`: a pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size (Definition 176) to the message that is to be signed.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the 64-byte signature. This function returns if the public key cannot be found in

the key store.

17.1.4. `ext_crypto_ed25519_verify`

Verifies an *ed25519* signature.

17.1.4.1. Version 1 - Prototype

```
(func $ext_crypto_ed25519_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or a value equal to *0* if otherwise.

17.1.5. `ext_crypto_ed25519_batch_verify`

Registers a ed25519 signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section 17.1.20). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section 17.1.21). If batch verification is not enabled, the signature is verified immediately.

17.1.5.1. Version 1

```
(func $ext_crypto_ed25519_batch_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or batched or a value equal *0* to if otherwise.

17.1.6. `ext_crypto_sr25519_public_keys`

Returns all *sr25519* public keys for the given key id from the keystore.

17.1.6.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_public_keys_version_1
    (param $key_type_id i32) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key type identifier (Definition 179).
- `return`: a pointer-size (Definition 176) to the SCALE encoded 256-bit public keys.

17.1.7. `ext_crypto_sr25519_generate`

Generates an *sr25519* key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

17.1.7.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_generate_version_1
    (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key identifier (Definition 179).
- `seed`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the BIP-39 seed which must be valid UTF8.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit public key.

17.1.8. `ext_crypto_sr25519_sign`

Signs the given message with the *sr25519* key that corresponds to the given public key and key type in the keystore.

17.1.8.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_sign_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key identifier (Definition 179).
- `key`: a pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size (Definition 176) to the message that is to be signed.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the 64-byte signature. This function returns *None* if the public key cannot be

found in the key store.

17.1.9. `ext_crypto_sr25519_verify`

Verifies an sr25519 signature.

17.1.9.1. Version 1 - Prototype

```
(func $ext_crypto_sr25519_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or a value equal to *0* if otherwise.

17.1.9.2. Version 2 - Prototype

```
(func $ext_crypto_sr25519_verify_version_2
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or a value equal to *0* if otherwise.

17.1.10. `ext_crypto_sr25519_batch_verify`

Registers a sr25519 signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section 17.1.20). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section 17.1.21). If batch verification is not enabled, the signature is verified immediately.

17.1.10.1. Version 1

```
(func $ext_crypto_sr25519_batch_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or batched or a value equal *0* to if otherwise.

17.1.11. `ext_crypto_ecdsa_public_keys`

Returns all *ecdsa* public keys for the given key id from the keystore.

17.1.11.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_public_key_version_1
    (param $key_type_id i64) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key type identifier (Definition 179).
- `return`: a pointer-size (Definition 176) to the SCALE encoded 33-byte compressed public keys.

17.1.12. `ext_crypto_ecdsa_generate`

Generates an *ecdsa* key for the given key type using an optional BIP-39 seed and stores it in the keystore.

Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

17.1.12.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_generate_version_1
    (param $key_type_id i32) (param $seed i64) (return i32))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key identifier (Definition 179).
- `seed`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the BIP-39 seed which must be valid UTF8.
- `return`: a pointer (Definition 175) to the buffer containing the 33-byte compressed public key.

17.1.13. `ext_crypto_ecdsa_sign`

Signs the hash of the given message with the *ecdsa* key that corresponds to the given public key and key type in the keystore.

17.1.13.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_sign_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments

- `key_type_id`: a pointer (Definition 175) to the key identifier (Definition 179).
- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size (Definition 176) to the message that is to be signed.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID. This function returns if the public key cannot be found in the key store.

17.1.14. `ext_crypto_ecdsa_sign_prehashed`

Signs the prehashed message with the *ecdsa* key that corresponds to the given public key and key type in the keystore.

17.1.14.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_sign_prehashed_version_1
    (param $key_type_id i32) (param $key i32) (param $msg i64) (return i64))
```

Arguments

- `key_type_id`: a pointer-size (Definition 175) to the key identifier (Definition 179).
- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size (Definition 176) to the message that is to be signed.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID. This function returns if the public key cannot be found in the key store.

17.1.15. `ext_crypto_ecdsa_verify`

Verifies the hash of the given message against a ECDSA signature.

17.1.15.1. Version 1 - Prototype

This function allows the verification of non-standard, overflowing ECDSA signatures, an implemenation specific mechanism of the Rust `libsecp256k1` library, specifically the `parse_overflowing` function.

```
(func $ext_crypto_ecdsa_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `return`: a i32 integer value equal *1* to if the signature is valid or a value equal to *0* if otherwise.

17.1.15.2. Version 2 - Prototype

Does not allow the verification of non-standard, overflowing ECDSA signatures.

```
(func $ext_crypto_ecdsa_verify_version_2
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `return`: a i32 integer value equal *1* to if the signature is valid or a value equal to *0* if otherwise.

17.1.16. `ext_crypto_ecdsa_verify_prehashed`

Verifies the prehashed message against a ECDSA signature.

17.1.16.1. Version 1 - Prototype

```
(func $ext_crypto_ecdsa_verify_prehashed_version_1
    (param $sig i32) (param $msg i32) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature. The signature is 65-bytes in size, where the first 512-bits represent the signature and the other 8 bits represent the recovery ID.
- `msg`: a pointer to the 32-bit prehashed message to be verified.

- `key`: a pointer to the 33-byte compressed public key.
- `return`: a i32 integer value equal *1* to if the signature is valid or a value equal to *0* if otherwise.

17.1.17. `ext_crypto_ecdsa_batch_verify`

Registers a ECDSA signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section 17.1.20). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section 17.1.21). If batch verification is not enabled, the signature is verified immediately.

17.1.17.1. Version 1

```
(func $ext_crypto_ecdsa_batch_verify_version_1
    (param $sig i32) (param $msg i64) (param $key i32) (return i32))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 176) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: a i32 integer value equal to *1* if the signature is valid or batched or a value equal *0* to if otherwise.

17.1.18. `ext_crypto_secp256k1_ecdsa_recover`

Verify and recover a *secp256k1* ECDSA signature.

17.1.18.1. Version 1 - Prototype

This function can handle non-standard, overflowing ECDSA signatures, an implemenation specific mechanism of the Rust `libsecp256k1` library, specifically the `parse_overflowing` function.

```
(func $ext_crypto_secp256k1_ecdsa_recover_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature in RSV format. V should be either or .
- `msg`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Result* (Definition 165). On success it contains the 64-byte recovered public key or an error type (Definition 180) on failure.

17.1.18.2. Version 2 - Prototype

Does not handle non-standard, overflowing ECDSA signatures.

```
(func $ext_crypto_secp256k1_ecdsa_recover_version_2
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature in RSV format. V should be either or .
- `msg`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Result* (Definition 165). On success it contains the 64-byte recovered public key or an error type (Definition 180) on failure.

17.1.19. `ext_crypto_secp256k1_ecdsa_recover_compressed`

Verify and recover a *secp256k1* ECDSA signature.

17.1.19.1. Version 1 - Prototype

This function can handle non-standard, overflowing ECDSA signatures, an implemenation specific mechanism of the Rust `libsecp256k1` library, specifically the `parse_overflowing` function.

```
(func $ext_crypto_secp256k1_ecdsa_recover_compressed_version_1
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 176) to the SCALE encoded `Result` value (Definition 165). On success it contains the 33-byte recovered public key in compressed form on success or an error type (Definition 180) on failure.

17.1.19.2. Version 2 - Prototype

Does not handle non-standard, overflowing ECDSA signatures.

```
(func $ext_crypto_secp256k1_ecdsa_recover_compressed_version_2
    (param $sig i32) (param $msg i32) (return i64))
```

Arguments

- `sig`: a pointer (Definition 175) to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a pointer (Definition 175) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 176) to the SCALE encoded `Result` value (Definition 165). On success it contains the 33-byte recovered public key in compressed form on success or an error type (Definition 180) on failure.

17.1.20. `ext_crypto_start_batch_verify`

Starts the verification extension. The extension is a separate background process and is used to parallel-verify signatures which are pushed to the batch with `ext_crypto_ed25519_batch_verify` (Section 17.1.5), `ext_crypto_sr25519_batch_verify` (Section 17.1.10) or `ext_crypto_ecdsa_batch_verify` (Section 17.1.17). Verification will start immediately and the Runtime can retrieve the result when calling `ext_crypto_finish_batch_verify` (Section 17.1.21).

17.1.20.1. Version 1 - Prototype

```
(func $ext_crypto_start_batch_verify_version_1)
```

Arguments

- None.

17.1.21. `ext_crypto_finish_batch_verify`

Finish verifying the batch of signatures since the last call to this function. Blocks until all the signatures are verified.

Panics if `ext_crypto_start_batch_verify` (Section 17.1.20) was not called.

17.1.21.1. Version 1 - Prototype

```
(func $ext_crypto_finish_batch_verify_version_1
    (return i32))
```

Arguments

- `return`: an i32 integer value equal to *1* if all the signatures are valid or a value equal to *0* if one or more of the signatures are invalid.

Chapter 18. Hashing

Interface that provides functions for hashing with different algorithms.

18.1. Functions

18.1.1. `ext_hashing_keccak_256`

Conducts a 256-bit Keccak hash.

18.1.1.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit hash result.

18.1.2. `ext_hashing_keccak_512`

Conducts a 512-bit Keccak hash.

18.1.2.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_512_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 512-bit hash result.

18.1.3. `ext_hashing_sha2_256`

Conducts a 256-bit Sha2 hash.

18.1.3.1. Version 1 - Prototype

```
(func $ext_hashing_sha2_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.

- `return`: a pointer (Definition 175) to the buffer containing the 256-bit hash result.

18.1.4. `ext_hashing_blake2_128`

Conducts a 128-bit Blake2 hash.

18.1.4.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_128_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 128-bit hash result.

18.1.5. `ext_hashing_blake2_256`

Conducts a 256-bit Blake2 hash.

18.1.5.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit hash result.

18.1.6. `ext_hashing_twox_64`

Conducts a 64-bit xxHash hash.

18.1.6.1. Version 1 - Prototype

```
(func $ext_hashing_twox_64_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 64-bit hash result.

18.1.7. `ext_hashing_twox_128`

Conducts a 128-bit xxHash hash.

18.1.7.1. Version 1 - Prototype

```
(func $ext_hashing_twox_128
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 128-bit hash result.

18.1.8. `ext_hashing_twox_256`

Conducts a 256-bit xxHash hash.

18.1.8.1. Version 1 - Prototype

```
(func $ext_hashing_twox_256
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the data to be hashed.
- `return`: a pointer (Definition 175) to the buffer containing the 256-bit hash result.

Chapter 19. Offchain

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 Definition 181 and Definition 182.

Definition 181. Persisted Storage

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 182. Local Storage

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 183. HTTP Status Code

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.

Definition 184. HTTP Error

HTTP error, E, is a varying data type (Definition 162) and specifies the error types of certain HTTP functions. Following values are possible:

$$E = \begin{cases} 0 & \text{The deadile was reached} \\ 1 & \text{There was an IO error while processing the request} \\ 2 & \text{The Id of the request is invalid} \end{cases}$$

19.1. Functions

19.1.1. `ext_offchain_is_validator`

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

19.1.1.1. Version 1 - Prototype

```
(func $ext_offchain_is_validator_version_1 (return i32))
```

Arguments

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

19.1.2. `ext_offchain_submit_transaction`

Given a SCALE encoded extrinsic, this function submits the extrinsic to the Host's transaction pool, ready to be propagated to remote peers.

19.1.2.1. Version 1 - Prototype

```
(func $ext_offchain_submit_transaction_version_1
    (param $data i64) (return i64))
```

Arguments

- `data`: a pointer-size (Definition 176) to the byte array storing the encoded extrinsic.
- `return`: a pointer-size (Definition 176) to the SCALE encoded *Result* value (Definition 165). Neither on success or failure is there any additional data provided. The cause of a failure is implementation specific.

19.1.3. `ext_offchain_network_state`

Returns the SCALE encoded, opaque information about the local node's network state.

Definition 185. Opaque Network State

The **Opaque network state structure**, S, is a SCALE encoded blob holding information about the the *libp2p PeerId*, P_{id}, of the local node and a list of *libp2p Multiaddresses*, $(M_0, \text{ellipsis}\, M_n)$, the node knows it can be reached at:

$$S = (P_{\text{id}}, (M_0, \text{ellipsis}\, M_n))$$

where

$$P_{\text{id}} = (b_0, \text{ellipsis}\, b_n) M = (b_0, \text{ellipsis}\, b_n)$$

The information contained in this structure is naturally opaque to the caller of this function.

19.1.3.1. Version 1 - Prototype

```
(func $ext_offchain_network_state_version_1 (result i64))
```

Arguments

- `result`: a pointer-size (Definition 176) to the SCALE encoded `Result` value (Definition 165). On success it contains the *Opaque network state* structure (Definition 185). On failure, an empty value is yielded where its cause is implementation specific.

19.1.4. `ext_offchain_timestamp`

Returns the current timestamp.

19.1.4.1. Version 1 - Prototype

```
(func $ext_offchain_timestamp_version_1 (result u64))
```

Arguments

- `result`: an u64 integer indicating the current UNIX timestamp (Definition 4).

19.1.5. `ext_offchain_sleep_until`

Pause the execution until the `deadline` is reached.

19.1.5.1. Version 1 - Prototype

```
(func $ext_offchain_sleep_until_version_1 (param $deadline u64))
```

Arguments

- `deadline`: an u64 integer specifying the UNIX timestamp (Definition 4).

19.1.6. `ext_offchain_random_seed`

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

19.1.6.1. Version 1 - Prototype

```
(func $ext_offchain_random_seed_version_1 (result i32))
```

Arguments

- `result`: a pointer (Definition 175) to the buffer containing the 256-bit seed.

19.1.7. `ext_offchain_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.

19.1.7.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_set_version_1
    (param $kind i32) (param $key i64) (param $value i64))
```

Arguments

- `kind`: an i32 integer indicating the storage kind. A value equal to *1* is used for a persistent storage (Definition 181) and a value equal to *2* for local storage (Definition 182).
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.

19.1.8. `ext_offchain_local_storage_clear`

Remove a value from the local storage.

19.1.8.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_clear_version_1
    (param $kind i32) (param $key i64))
```

Arguments

- `kind`: an i32 integer indicating the storage kind. A value equal to *1* is used for a persistent storage (Definition 181) and a value equal to *2* for local storage (Definition 182).
- `key`: a pointer-size (Definition 176) to the key.

19.1.9. `ext_offchain_local_storage_compare_and_set`

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

19.1.9.1. Version 1 - Prototype

```
(fund $ext_offchain_local_storage_compare_and_set_version_1
    (param $kind i32) (param $key i64) (param $old_value i64)
    (param $new_value i64) (result i32))
```

Arguments

- `kind`: an i32 integer indicating the storage kind. A value equal to *1* is used for a persistent storage (Definition 181) and a value equal to *2* for local storage (Definition 182).
- `key`: a pointer-size (Definition 176) to the key.
- `old_value`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the old key.
- `new_value`: a pointer-size (Definition 176) to the new value.
- `result`: an i32 integer equal to *1* if the new value has been set or a value equal to *0* if otherwise.

19.1.10. `ext_offchain_local_storage_get`

Gets a value from the local storage.

19.1.10.1. Version 1 - Prototype

```
(func $ext_offchain_local_storage_get_version_1
    (param $kind i32) (param $key i64) (result i64))
```

Arguments

- `kind`: an i32 integer indicating the storage kind. A value equal to *1* is used for a persistent storage (Definition 181) and a value equal to *2* for local storage (Definition 182).
- `key`: a pointer-size (Definition 176) to the key.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the value or the corresponding key.

19.1.11. `ext_offchain_http_request_start`

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

19.1.11.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_start_version_1
  (param $method i64) (param $uri i64) (param $meta i64) (result i64))
```

Arguments

- `method`: a pointer-size (Definition 176) to the HTTP method. Possible values are “GET” and “POST”.
- `uri`: a pointer-size (Definition 176) to the URI.
- `meta`: a future-reserved field containing additional, SCALE encoded parameters. Currently, an empty array should be passed.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Result* value (Definition 165) containing the i16 ID of the newly started request. On failure no additionally data is provided. The cause of failure is implementation specific.

19.1.12. `ext_offchain_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).

19.1.12.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_add_header_version_1
    (param $request_id i32) (param $name i64) (param $value i64) (result i64))
```

Arguments

- `request_id`: an i32 integer indicating the ID of the started request.
- `name`: a pointer-size (Definition 176) to the HTTP header name.
- `value`: a pointer-size (Definition 176) to the HTTP header value.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Result* value (Definition 165). Neither on success or failure is there any additional data provided. The cause of failure is implementation specific.

19.1.13. `ext_offchain_http_request_write_body`

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

19.1.13.1. Version 1 - Prototype

```
(func $ext_offchain_http_request_write_body_version_1
    (param $request_id i32) (param $chunk i64) (param $deadline i64) (result i64))
```

Arguments

- `request_id`: an i32 integer indicating the ID of the started request.
- `chunk`: a pointer-size (Definition 176) to the chunk of bytes. Writing an empty chunk finalizes the request.
- `deadline`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the UNIX timestamp (Definition 4). Passing *None* blocks indefinitely.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Result* value (Definition 165). On success, no additional data is provided. On error it contains the HTTP error type (Definition 184).

19.1.14. `ext_offchain_http_response_wait`

Returns an array of request statuses (the length is the same as IDs). Note that if deadline is not provided the method will block indefinitely, otherwise unready responses will produce DeadlineReached status.

19.1.14.1. Version 1 - Prototype

```
(func $ext_offchain_http_response_wait_version_1
    (param $ids i64) (param $deadline i64) (result i64))
```

Arguments

- `ids`: a pointer-size (Definition 176) to the SCALE encoded array of started request IDs.
- `deadline`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the UNIX timestamp (Definition 4). Passing None blocks indefinitely.
- `result`: a pointer-size (Definition 176) to the SCALE encoded array of request statuses ([defn-http-status-codes]).

19.1.15. `ext_offchain_http_response_headers`

Read all HTTP response headers. Returns an array of key/value pairs. Response headers must be read before the response body.

19.1.15.1. Version 1 - Prototype

```
(func $ext_offchain_http_response_headers_version_1
    (param $request_id i32) (result i64))
```

Arguments

- `request_id`: an i32 integer indicating the ID of the started request.
- `result`: a pointer-size (Definition 176) to a SCALE encoded array of key/value pairs.

19.1.16. `ext_offchain_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.

19.1.16.1. Version 1 - Prototype

```
(func $ext_offchain_http_response_read_body_version_1
    (param $request_id i32) (param $buffer i64) (param $deadline i64) (result i64))
```

Arguments

- `request_id`: an i32 integer indicating the ID of the started request.
- `buffer`: a pointer-size (Definition 176) to the buffer where the body gets written to.
- `deadline`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the UNIX timestamp (Definition 4). Passing *None* will block indefinitely.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Result* value (Definition 165). On success it contains an i32 integer specifying the number of bytes written or a HTTP error type (Definition 184) on failure.

Chapter 20. Trie

Interface that provides trie related functionality.

20.1. Functions

20.1.1. `ext_trie_blake2_256_root`

Compute a 256-bit Blake2 trie root formed from the iterated items.

20.1.1.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs (tuples).
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root.

20.1.1.2. Version 2 - Prototype

```
(func $ext_trie_blake2_256_root_version_2
    (param $data i64) (param $version i32)
    (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs (tuples).
- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root.

20.1.2. `ext_trie_blake2_256_ordered_root`

Compute a 256-bit Blake2 trie root formed from the enumerated items.

20.1.2.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_ordered_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers (Definition 171).
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root result.

20.1.2.2. Version 2 - Prototype

```
(func $ext_trie_blake2_256_ordered_root_version_2
    (param $data i64) (param $version i32)
    (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers (Definition 171).
- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root result.

20.1.3. `ext_trie_keccak_256_root`

Compute a 256-bit Keccak trie root formed from the iterated items.

20.1.3.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs.
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root.

20.1.3.2. Version 2 - Prototype

```
(func $ext_trie_keccak_256_root_version_2
    (param $data i64) (param $version i32)
    (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the iterated items from which the trie root gets formed. The items consist of a SCALE encoded array containing arbitrary key/value pairs.

- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root.

20.1.4. `ext_trie_keccak_256_ordered_root`

Compute a 256-bit Keccak trie root formed from the enumerated items.

20.1.4.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_ordered_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers (Definition 171).
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root result.

20.1.4.2. Version 2 - Prototype

```
(func $ext_trie_keccak_256_ordered_root_version_2
    (param $data i64) (param $version i32)
    (result i32))
```

Arguments

- `data`: a pointer-size (Definition 176) to the enumerated items from which the trie root gets formed. The items consist of a SCALE encoded array containing only values, where the corresponding key of each value is the index of the item in the array, starting at 0. The keys are compact encoded integers (Definition 171).
- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `result`: a pointer (Definition 175) to the buffer containing the 256-bit trie root result.

20.1.5. `ext_trie_blake2_256_verify_proof`

Verifies a key/value pair against a Blake2 256-bit merkle root.

20.1.5.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_verify_proof_version_1
    (param $root i32) (param $proof i64)
    (param $key i64) (param $value i64)
    (result i32))
```

Arguments

- `root`: a pointer to the 256-bit merkle root.
- `proof`: a pointer-size (Definition 176) to an array containing the node proofs.
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

20.1.5.2. Version 2 - Prototype

```
(func $ext_trie_blake2_256_verify_proof_version_2
    (param $root i32) (param $proof i64)
    (param $key i64) (param $value i64)
    (param $version i32) (result i32))
```

Arguments

- `root`: a pointer to the 256-bit merkle root.
- `proof`: a pointer-size (Definition 176) to an array containing the node proofs.
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.
- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

20.1.6. `ext_trie_keccak_256_verify_proof`

Verifies a key/value pair against a Keccak 256-bit merkle root.

20.1.6.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_verify_proof_version_1
    (param $root i32) (param $proof i64)
    (param $key i64) (param $value i64)
    (result i32))
```

Arguments

- `root`: a pointer to the 256-bit merkle root.
- `proof`: a pointer-size (Definition 176) to an array containing the node proofs.
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if

otherwise.

20.1.6.2. Version 2 - Prototype

```
(func $ext_trie_keccak_256_verify_proof_version_2
    (param $root i32) (param $proof i64)
    (param $key i64) (param $value i64)
    (param $version i32) (result i32))
```

Arguments

- `root`: a pointer to the 256-bit merkle root.
- `proof`: a pointer-size (Definition 176) to an array containing the node proofs.
- `key`: a pointer-size (Definition 176) to the key.
- `value`: a pointer-size (Definition 176) to the value.
- `version`: a pointer-size (Definition 175) to the state version Definition 173.
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

Chapter 21. Miscellaneous

Interface that provides miscellaneous functions for communicating between the runtime and the node.

21.1. Functions

21.1.1. `ext_misc_print_num`

Print a number.

21.1.1.1. Version 1 - Prototype

```
(func $ext_misc_print_num_version_1 (param $value i64))
```

Arguments

- `value`: the number to be printed.

21.1.2. `ext_misc_print_utf8`

Print a valid UTF8 encoded buffer.

21.1.2.1. Version 1 - Prototype

```
(func $ext_misc_print_utf8_version_1 (param $data i64))
```

Arguments:

- : a pointer-size (Definition 176) to the valid buffer to be printed.

21.1.3. `ext_misc_print_hex`

Print any buffer in hexadecimal representation.

21.1.3.1. Version 1 - Prototype

```
(func $ext_misc_print_hex_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size (Definition 176) to the buffer to be printed.

21.1.4. `ext_misc_runtime_version`

Extract the Runtime version of the given Wasm blob by calling `Core_version` (Section 26.1.1).

Returns the SCALE encoded runtime version or *None* (Definition 164) if the call fails. This function gets primarily used when upgrading Runtimes.

Calling this function is very expensive and should only be done very occasionally. For getting the runtime version, it requires instantiating the Wasm blob (Section 3.1.1) and calling the `Core_version` function (Section 26.1.1) in this blob.

21.1.4.1. Version 1 - Prototype

```
(func $ext_misc_runtime_version_version_1 (param $data i64) (result i64))
```

Arguments

- `data`: a pointer-size (Definition 176) to the Wasm blob.
- `result`: a pointer-size (Definition 176) to the SCALE encoded *Option* value (Definition 164) containing the Runtime version of the given Wasm blob which is encoded as a byte array.

Chapter 22. Allocator

The Polkadot Runtime does not include a memory allocator and relies on the Host API for all heap allocations. The beginning of this heap is marked by the `__heap_base` symbol exported by the Polkadot Runtime. No memory should be allocated below that address, to avoid clashes with the stack and data section. The same allocator made accessible by this Host API should be used for any other WASM memory allocations and deallocations outside the runtime e.g. when passing the SCALE-encoded parameters to Runtime API calls.

22.1. Functions

22.1.1. `ext_allocator_malloc`

Allocates the given number of bytes and returns the pointer to that memory location.

22.1.1.1. Version 1 - Prototype

```
(func $ext_allocator_malloc_version_1 (param $size i32) (result i32))
```

Arguments

- `size`: the size of the buffer to be allocated.
- `result`: a pointer (Definition 175) to the allocated buffer.

22.1.2. `ext_allocator_free`

Free the given pointer.

22.1.2.1. Version 1 - Prototype

```
(func $ext_allocator_free_version_1 (param $ptr i32))
```

Arguments

- `ptr`: a pointer (Definition 175) to the memory buffer to be freed.

Chapter 23. Logging

Interface that provides functions for logging from within the runtime.

Definition 186. Log Level

The **Log Level**, L, is a varying data type (Definition 162) and implies the emergency of the log. Possible log levels and the corresponding identifier is as follows:

$$L = \begin{cases} 0 & \text{Error} = 1 \\ 1 & \text{Warn} = 2 \\ 2 & \text{Info} = 3 \\ 3 & \text{Debug} = 4 \\ 4 & \text{Trace} = 5 \end{cases}$$

23.1. Functions

23.1.1. `ext_logging_log`

Request to print a log message on the host. Note that this will be only displayed if the host is enabled to display log messages with given level and target.

23.1.1.1. Version 1 - Prototype

```
(func $ext_logging_log_version_1
    (param $level i32) (param $target i64) (param $message i64))
```

Arguments

- `level`: the log level (Definition 186).
- `target`: a pointer-size (Definition 176) to the string which contains the path, module or location from where the log was executed.
- `message`: a pointer-size (Definition 176) to the log message.

Runtime API

Description of how to interact with the Runtime through its exported functions

Chapter 24. General Information

The Polkadot Host assumes that at least the constants and functions described in this Chapter are implemented in the Runtime Wasm blob.

It should be noted that the API can change through the Runtime updates. Therefore, a host should check the API versions of each module returned in the `api` field by `Core_version` (Section 26.1.1) after every Runtime upgrade and warn if an updated API is encountered and that this might require an update of the host.

24.1. JSON-RPC API for external services

Polkadot Host implementers are encouraged to implement an API in order for external, third-party services to interact with the node. The JSON-RPC Interface for Polkadot Nodes (PSP6) is a Polkadot Standard Proposal for such an API and makes it easier to integrate the node with existing tools available in the Polkadot ecosystem, such as polkadot.js.org. The Runtime API has a few modules designed specifically for use in the official RPC API.

Chapter 25. Runtime Constants

25.1. `__heap_base`

This constant indicates the beginning of the heap in memory. The space below is reserved for the stack and the data section. For more details please refer to [sect-ext-allocator-api].

Chapter 26. Runtime Functions

In this section, we describe all Runtime API functions alongside their arguments and the return values. The functions are organized into modules with each being versioned independently.

Definition 187. Runtime API Call Convention

The **Runtime API Call Convention** describes that all functions receive and return SCALE-encoded data and as a result have the following prototype signature:

```
(func $generic_runtime_entry
  (param $ptr i32) (parm $len i32) (result i64))
```

where `ptr` points to the SCALE encoded tuple of the parameters passed to the function and `len` is the length of this data, while `result` is a pointer-size (Definition Definition 176) to the SCALE-encoded return data.

See Section 3.1.2 for more information about the behavior of the Wasm Runtime. Also note that any state changes created by calling any of the Runtime functions are not necessarily to be persisted after the call is ended. See Section 3.1.2.4 for more information.

26.1. Core Module (Version 3)

26.1.1. `Core_version`

Returns the version identifiers of the Runtime. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section 24.1.

Arguments

- None

Return

- A datastructure of the following format:

Table 6. Details of the version that the data type returns from the Runtime function.

Name	Type	Description
`spec_name`	String	Runtime identifier
`impl_name`	String	Name of the implementation (e.g. C++)
`authoring_version`	Unsigned 32-bit integer	Version of the authorship interface
`spec_version`	Unsigned 32-bit integer	Version of the Runtime specification

Name	Type	Description
`impl_version`	Unsigned 32-bit integer	Version of the Runtime implementation
`apis`	ApiVersions (Definition 188)	List of supported APIs along with their version
`transaction_version`	Unsigned 32-bit integer	Version of the transaction format
`state_version`	Unsigned 32-bit integer	Version of the trie format

Definition 188. ApiVersions

ApiVersions is a specialized type for the (Section 26.1.1) function entry. It represents an array of tuples, where the first value of the tuple is an array of 8-bytes containing the Blake2b hash of the API name. The second value of the tuple is the version number of the corresponding API.

$$\begin{aligned} \text{ApiVersions} &:= (T_0, ..., T_n) \\ T &:= ((b_0, ..., b_7), \text{UINT32}) \end{aligned}$$

Requires `Core_initialize_block` to be called beforehand.

26.1.2. `Core_execute_block`

This function executes a full block and all its extrinsics and updates 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 12, 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. State changes resulted from calling this function are usually meant to persist when the block is imported successfully.

Additionally, the seal digest in the block header, as described in Definition 29, must be removed by the Polkadot host before submitting the block.

Arguments

- A block represented as a tuple consisting of a block header, as described in Definition 28, and the block body, as described in Definition 31.

Return

- None.

26.1.3. `Core_initialize_block`

Sets up the environment required for building a new block as described in Algorithm 12.

Arguments

- The header of the new block as defined in Definition 28. The values H_r, H_e and H_d are left empty.

Return

- None.

26.2. Metadata Module (Version 1)

26.2.1. `Metadata_metadata`

Returns native Runtime metadata in an opaque form. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section 24.1. and returns all the information necessary to build valid transactions.

Arguments

- None.

Return

- A byte array of varying size containing the metadata in an opaque form.

26.3. BlockBuilder Module (Version 4)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.3.1. `BlockBuilder_apply_extrinsic`

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

Arguments

- A byte array of varying size containing the opaque extrinsic.

Return

- Returns the varying datatype *ApplyExtrinsicResult* as defined in Definition 189. This structure lets the block builder know whether an extrinsic should be included into the block or rejected.

Definition 189. ApplyExtrinsicResult

ApplyExtrinsicResult is a varying data type as defined in Definition 165. This structure can contain multiple nested structures, indicating either module dispatch outcomes or transaction invalidity errors.

Table 7. Possible values of varying data type ApplyExtrinsicResult.

Id	Description	Type
0	Outcome of dispatching the extrinsic.	*DispatchOutcome* (Definition 190)
1	Possible errors while checking the validity of a transaction.	*TransactionValidityError* (Definition 193)

As long as a *DispatchOutcome* (Definition 190) is returned, the extrinsic is always included in the block, even if the outcome is a dispatch error. Dispatch errors do not invalidate the block and all state changes are persisted.

Definition 190. DispatchOutcome

DispatchOutcome is the varying data type as defined in Definition 165.

Table 8. Possible values of varying data type DispatchOutcome.

Id	Description	Type
0	Extrinsic is valid and was submitted successfully.	None
1	Possible errors while dispatching the extrinsic.	*DispatchError* (Definition 191)

Definition 191. DispatchError

DispatchError is a varying data type as defined in Definition 162. Indicates various reasons why a dispatch call failed.

Table 9. Possible values of varying data type DispatchError.

Id	Description	Type
0	Some unknown error occurred.	SCALE encoded byte array containing a valid UTF-8 sequence.
1	Failed to lookup some data.	None
2	A bad origin.	None
3	A custom error in a module.	*CustomModuleError* (Definition 192)

Definition 192. CustomModuleError

CustomModuleError is a tuple appended after a possible error in as defined in Definition 191.

Table 10. Possible values of varying data type CustomModuleError.

Name	Description	Type
Index	Module index matching the metadata module index.	Unsigned 8-bit integer.
Error	Module specific error value.	Unsigned 8-bit integer
Message	Optional error message.	Varying data type *Option* (Definition 164). The optional value is a SCALE encoded byte array containing a valid UTF-8 sequence.

Whenever *TransactionValidityError* (Definition 193) is returned, the contained error type will indicate whether an extrinsic should be outright rejected or requested for a later block. This behavior is clarified further in Definition 194 and respectively Definition 195.

Definition 193. TransactionValidityError

TransactionValidityError is a varying data type as defined in Definition 162. It indicates possible errors that can occur while checking the validity of a transaction.

Table 11. Possible values of varying data type TransactionValidityError.

Id	Description	Type
0	Transaction is invalid.	*InvalidTransaction* (Definition 194)
1	Transaction validity can’t be determined.	*UnknownTransaction* (Definition 195)

Definition 194. InvalidTransaction

InvalidTransaction is a varying data type as defined in Definition 162 and specifies the invalidity of the transaction in more detail.

Table 12. Possible values of varying data type InvalidTransaction.

Id	Description	Type	Reject
0	Call of the transaction is not expected.	None	Yes
1	General error to do with the inability to pay some fees (e.g. account balance too low).	None	Yes
2	General error to do with the transaction not yet being valid (e.g. nonce too high).	None	No
3	General error to do with the transaction being outdated (e.g. nonce too low).	None	Yes
4	General error to do with the transactions' proof (e.g. signature)	None	Yes
5	The transaction birth block is ancient.	None	Yes
6	The transaction would exhaust the resources of the current block.	None	No
7	Some unknown error occurred.	Unsigned 8-bit integer	Yes
8	An extrinsic with mandatory dispatch resulted in an error.	None	Yes
9	A transaction with a mandatory dispatch (only inherents are allowed to have mandatory dispatch).	None	Yes

Definition 195. UnknownTransaction

UnknownTransaction is a varying data type as defined in Definition 162 and specifies the unknown invalidity of the transaction in more detail.

Table 13. Possible values of varying data type UnknownTransaction.

Id	Description	Type	Reject
0	Could not lookup some information that is required to validate the transaction.	None	Yes
1	No validator found for the given unsigned transaction.	None	Yes
2	Any other custom unknown validity that is not covered by this type.	Unsigned 8-bit integer	Yes

26.3.2. 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. State changes resulting from calling this function are usually meant to persist upon successful execution of the function and appending of the block to the chain.

Arguments

- None.

Return

- The header of the new block as defined in Definition 28.

26.3.3. BlockBuilder_inherent_extrinisics:

Generates the inherent extrinsics, which are explained in more detail in Section 3.2.3. This function takes a SCALE-encoded hash table as defined in Definition 166 and returns an array of extrinsics. The Polkadot Host must submit each of those to the BlockBuilder_apply_extrinsic, described in Section 26.3.1. This procedure is outlined in Algorithm 12.

Arguments

- A Inherents-Data structure as defined in Section 3.2.3.1.

Return

- A byte array of varying size containing extrinisics. Each extrinsic is a byte array of varying size.

26.3.4. BlockBuilder_check_inherents

Checks whether the provided inherent is valid. This function can be used by the Polkadot Host when deemed appropriate, e.g. during the block-building process.

Arguments

- A block represented as a tuple consisting of a block header as described in Definition 28 and the block body as described in Definition 31.
- A Inherents-Data structure as defined in Section 3.2.3.1.

Return

- A data structure of the following format:

$$(o, f_e, e)$$

  where

  - o is a boolean indicating whether the check was successful.
  - f_e is a boolean indicating whether a fatal error was encountered.
  - e is a Inherents-Data structure as defined in Section 3.2.3.1 containing any errors created by this Runtime function.

26.3.5. `BlockBuilder_random_seed`

Generates a random seed.

Arguments

- None

Return

- A 32-byte array containing the random seed.

26.4. TaggedTransactionQueue (Version 2)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.4.1. `TaggedTransactionQueue_validate_transaction`

This entry is invoked against extrinsics submitted through a transaction network message (Section 4.8.5) or by an offchain worker through the Host API (Section 19.1.2).

It indicates if the submitted blob represents a valid extrinsics, the order in which it should be applied and if it should be gossiped to other peers. Furthermore this function gets called internally when executing blocks with the runtime function as described in Section 26.1.2.

Arguments

- The source of the transaction as defined in Definition 196.
- A byte array that contains the transaction.

Definition 196. TransactionSource

TransactionSource is an enum describing the source of a transaction and can have one of the following values:

Table 14. The TransactionSource enum

Id	Name	Description
0	*InBlock*	Transaction is already included in a block.
1	*Local*	Transaction is coming from a local source, e.g. off-chain worker.
2	*External*	Transaction has been received externally, e.g. over the network.

Return

- This function returns a *Result* as defined in Definition 165 which contains the type *ValidTransaction* as defined in Definition 197 on success and the type *TransactionValidityError* as defined in Definition 193 on failure.

Definition 197. ValidTransaction

ValidTransaction is a tuple that contains information concerning a valid transaction.

Table 15. The tuple provided by in the case the transaction is judged to be valid.

Name	Description	Type
Priority	Determines the ordering of two transactions that have all their dependencies (required tags) are satisfied.	Unsigned 64bit integer
Requires	List of tags specifying extrinsics which should be applied before the current exrinsics can be applied.	Array containing inner arrays
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 containing inner arrays
Longevity	After this period, the transaction should be removed from the pool or revalidated.	Unsigned 64-bit integer
Propagate	A flag indicating if the transaction should be gossiped to other peers.	Boolean

If *Propagate* is set to `false` the transaction will still be considered for inclusion in blocks that are authored on the current node, but should not be gossiped to other peers.

If this function gets called by the Polkadot Host in order to validate a transaction received from peers, the Polkadot Host disregards and rewinds state changes resulting in such a call.

26.5. OffchainWorkerApi Module (Version 2)

Does not require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.5.1. `OffchainWorkerApi_offchain_worker`

Starts an off-chain worker and generates extrinsics. [To do: when is this called?]

Arguments

- The block header as defined in Definition 28.

Return

- None.

26.6. ParachainHost Module (Version 1)

26.6.1. ParachainHost_validators

Returns the validator set at the current state. The specified validators are responsible for backing parachains for the current state.

Arguments

- None.

Return

- An array of public keys representing the validators.

26.6.2. ParachainHost_validator_groups

Returns the validator groups (Section 6.8.7) used during the current session. The validators in the groups are referred to by the validator set Id (Definition 63).

Arguments

- None

Return

- An array of tuples, T, of the following format:

$$T = = (I, G) I = = (v_n, \text{ Unknown characterUnknown characterUnknown character} v_m) G = = (B_s, f, B_c)$$

where

- I is an array the validator set Ids (Definition 63).
- B_s indicates the block number where the session started.
- f indicates how often groups rotate. 0 means never.
- B_c indicates the current block number.

26.6.3. ParachainHost_availability_cores

Returns information on all availability cores (Section 6.8.6).

Arguments

- None

Return

- An array of core states, S, of the following format:

$$S = \begin{cases} 0 & -\&>; \ C_o \\ 1 & -\&>; \ C_s \ \ C_o = (n_u, B_o, B_t, n_t, b, G_i, C_h, C_d) C_s = (P_i d, C_i) \\ 2 & -\&>; \ \phi \end{cases}$$

where

- s specifies the core state. *0* indicates that the core is occupied, *1* implies it's currently free but scheduled and given the opportunity to occupy and *2* implies it's free and there's nothing scheduled.
- n_u is an *Option* value (Definition 164) which can contain a C_s value if the core was freed by the Runtime and indicates the assignment that is next scheduled on this core. An empty value indicates there is nothing scheduled.
- B_o indicates the relay chain block number at which the core got occupied.
- B_t indicates the relay chain block number the core will time-out at, if any.
- n_t is an *Option* value (Definition 164) which can contain a C_s value if the core is freed by a time-out and indicates the assignment that is next scheduled on this core. An empty value indicates there is nothing scheduled.
- b is a bitfield array (Section 6.8.12). A $> \frac{2}{3}$ majority of assigned validators voting with 1 values means that the core is available.
- G_i indicates the assigned validator group index (Section 6.8.7) is to distribute availability pieces of this candidate.
- C_h indicates the hash of the candidate occupying the core.
- C_d is the candidate descriptor (Definition 106).
- C_i is an *Option* value (Definition 164) which can contain the collators public key indicating who should author the block.

26.6.4. `ParachainHost_persisted_validation_data`

Returns the persisted validation data for the given parachain Id and a given occupied core assumption.

Arguments

- The parachain Id (Section 6.8.5).
- An occupied core assumption (Definition 198).

Return

- An *Option* value (Definition 164) which can contain the persisted validation data (Definition 199). The value is empty if the parachain Id is not registered or the core assumption is of index 2, meaning that the core was freed.

Definition 198. Occupied Core Assumption

A occupied core assumption is used for fetching certain pieces of information about a parachain by using the relay chain API. The assumption indicates how the Runtime API should compute the result. The assumptions, A, is a varying datatype of the following format:

$$A = = \begin{cases} 0 & -\&> ; \ \phi \\ 1 & -\&> ; \ \phi \\ 2 & -\&> ; \ \phi \end{cases}$$

where *0* indicates that the candidate occupying the core was made available and included to free the core, *1* indicates that it timed-out and freed the core without advancing the parachain and *2* indicates that the core was not occupied to begin with.

Definition 199. Persisted Validation Data

The persisted validation data provides information about how to create the inputs for the validation of a candidate by calling the Runtime. This information is derived from the parachain state and will vary from parachain to parachain, although some of the fields may be the same for every parachain. This validation data acts as a way to authorize the additional data (such as messages) the collator needs to pass to the validation function.

The persisted validation data, D_{pv}, is a datastructure of the following format:

$$D_{pv} = = (P_h, H_i, H_r, m_b)$$

where

- P_h is the parent head data (Section 6.8.4).
- H_i is the relay chain block number this is in the context of.
- H_r is the relay chain storage root this is in the context of.
- m_b is the maximum legal size of the PoV block, in bytes.

The persisted validation data is fetched via the Runtime API (Section 26.6.4).

26.6.5. `ParachainHost_check_validation_outputs`

Checks if the given validation outputs pass the acceptance criteria.

Arguments

- The parachain Id (Section 6.8.5).
- The candidate commitments (Definition 107).

Return

- A boolean indicating whether the candidate commitments pass the acceptance criteria.

26.6.6. ParachainHost_session_index_for_child

Returns the session index that is expected at the child of a block.

TODO clarify session index

Arguments

- None

Return

- A unsigned 32-bit integer representing the session index.

26.6.7. ParachainHost_validation_code

Fetches the validation code (Runtime) of a parachain by parachain Id.

Arguments

- The parachain Id (Section 6.8.5).
- The occupied core assumption (Definition 198).

Return

- An *Option* value (Definition 164) containing the full validation code in an byte array. This value is empty if the parachain Id cannot be found or the assumption is wrong.

26.6.8. ParachainHost_validation_code_by_hash

Returns the validation code (Runtime) of a parachain by its hash.

Arguments

- The hash value of the validation code.

Return

- An *Option* value (Definition 164) containing the full validation code in an byte array. This value is empty if the parachain Id cannot be found or the assumption is wrong.

26.6.9. ParachainHost_candidate_pending_availability

Returns the receipt of a candidate pending availability for any parachain assigned to an occupied availability core.

Arguments

- The parachain Id (Section 6.8.5).

Return

- An *Option* value (Definition 164) containing the committed candidate receipt (Definition 104). This value is empty if the given parachain Id is not assigned to an occupied availability cores.

26.6.10. ParachainHost_candidate_events

Returns an array of candidate events that occurred within the latest state.

Arguments

- None

Return

- An array of single candidate events, E, of the following format:

$$E == \begin{cases} 0 \quad -\&> ; \quad d \\ 1 \quad -\&> ; \quad d \\ 2 \quad -\&> ; \quad (C_r, h, I_c) \end{cases} \qquad d == (C_r, h, I_c, G_i)$$

  where

- E specifies the the event type of the candidate. *0* indicates that the candidate receipt was backed in the latest relay chain block, *1* indicates that it was included and became a parachain block at the latest relay chain block and *2* indicates that the candidate receipt was not made available and timed-out.
- C_r is the candidate receipt (Definition 104).
- h is the parachain head data (Section 6.8.4).
- I_c is the index of the availability core as can be retrieved in Section 26.6.3 that the candidate is occupying. If E is of variant 2, then this indicates the core index the candidate *was* occupying.
- G_i is the group index (Section 6.8.7) that is responsible of backing the candidate.

26.6.11. ParachainHost_session_info

Get the session info of the given session, if available.

Arguments

- The unsigned 32-bit integer indicating the session index.

Return

- An *Option* type (Definition 164) which can contain the session info structure, S, of the following format:

$$S == (A, D, K, G, \varepsilon, z, s, d, \chi, \sigma) A == (v_0, \text{Unknown character}\text{Unknown character}\text{Unknown character}v_m) D == (v_{a_0}, \text{Unknown character}\text{Unknown character}\text{Unknown character}v_m) K == (v_{a_0}, \text{Unknown character}\text{Unknown character}\text{Unknown character}v_m) G == (g_0, \text{Unknown character}\text{Unknown character}\text{Unknown character}g_m) \varepsilon == (A_0, \text{Unknown character}\text{Unknown character}\text{Unknown character}A_m)$$

  where

- A indicates the validators of the current session, in canonical order. There might be more validators in the current session than validators participating in parachain consensus, as returned by the Runtime API (Section 26.6.1).
- D indicates the validator authority discovery keys for the given session in canonical order. The first couple of validators are equal to the corresponding validators participating in the parachain consensus, as returned by the Runtime API (Section 26.6.1). The remaining authorities are not participating in the parachain consensus.

- K indicates the assignment keys for validators. There might be more authorities in the session that validators participating in parachain consensus, as returned by the Runtime API (Section 26.6.1).
- G indicates the validator groups in shuffled order.
- v_n is public key of the authority.
- A_n is the authority set Id (Definition 63).
- c is an unsigned 32-bit integer indicating the number of availability cores used by the protocol during the given session.
- z is an unsigned 32-bit integer indicating the zeroth delay tranche width.
- s is an unsigned 32-bit integer indicating the number of samples an assigned validator should do for approval voting.
- d is an unsigned 32-bit integer indicating the number of delay tranches in total.
- x is an unsigned 32-bit integer indicating how many BABE slots must pass before an assignment is considered a "no-show".
- a is an unsigned 32-bit integer indicating the number of validators needed to approve a block.

26.6.12. `ParachainHost_dmq_contents`

Returns all the pending inbound messages in the downward message queue for a given parachain.

Arguments

- The parachain Id (Section 6.8.5).

Return

- An array of inbound downward messages (Section 6.8.9).

26.6.13. `ParachainHost_inbound_hrmp_channels_contents`

Returns the contents of all channels addressed to the given recipient. Channels that have no messages in them are also included.

Arguments

- The parachain Id (Section 6.8.5).

Return

- An array of inbound HRMP messages (Section 6.8.11).

26.7. GrandpaApi Module (Version 2)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.7.1. GrandpaApi_grandpa_authorities

This entry fetches the list of GRANDPA authorities according to the genesis block and is used to initialize an authority list at genesis, defined in Definition 63. Any future authority changes get tracked via Runtime-to-consensus engine messages, as described in Section 5.1.2.

Arguments

- None.

Return

- An authority list as defined in Definition 63.

26.7.2. GrandpaApi_submit_report_equivocation_unsigned_extrinsic

A GRANDPA equivocation occurs when a validator votes for multiple blocks during one voting subround, as described further in Definition 88. The Polkadot Host is expected to identify equivocators and report those to the Runtime by calling this function.

Arguments

- The equivocation proof of the following format:

$$G_{\text{Ep}} = (\text{id}_V, e, r, A_{\text{id}}, B_h^1, B_n^1 A_{\text{sig}}^1, B_h^2, B_n^2, A_{\text{sig}}^2)$$
$$e = \begin{cases} 0 & \text{Equivocationatprevotestage} \\ 1 & \text{Equivocationatprecommitstage} \end{cases}$$

  where

- id_V is the authority set as defined in Definition 81.
- e indicates the stage at which the equivocation occurred.
- r is the round number the equivocation occurred.
- A_{id} is the public key of the equivocator.
- B_h^1 is the block hash of the first block the equivocator voted for.
- B_n^1 is the block number of the first block the equivocator voted for.
- A_{sig}^1 is the equivocators signature of the first vote.
- B_h^2 is the block hash of the second block the equivocator voted for.
- B_n^2 is the block number of the second block the equivocator voted for.
- A_{sig}^2 is the equivocators signature of the second vote.
- A proof of the key owner in an opaque form as described in Section 26.7.3.

Return

- A SCALE encoded *Option* as defined in Definition 164 containing an empty value on success.

26.7.3. GrandpaApi_generate_key_ownership_proof

Generates proof of the membership of a key owner in the specified block state. The returned value

is used to report equivocations as described in Section 26.7.2.

Arguments

- The authority set id as defined in Definition 81.
- The 256-bit public key of the authority.

Return

- A SCALE encoded *Option* as defined in Definition 164 containing the proof in an opaque form.

26.8. BabeApi Module (Version 2)

All calls in this module require `Core_initialized_block` (Section 26.1.3) to be called beforehand.

26.8.1. `BabeApi_configuration`

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

Arguments

- None.

Return

- A tuple containing configuration data used by the Babe consensus engine.

Table 16. The tuple provided by **BabeApi_configuration**.

Name	Description	Type
SlotDuration	The slot duration in milliseconds. Currently, only the value provided by this type at genesis will be used. Dynamic slot duration may be supported in the future.	Unsigned 64bit integer
EpochLength	The duration of epochs in slots.	Unsigned 64bit integer
Constant	A constant value that is used in the threshold calculation formula as defined in Definition 69.	Tuple containing two unsigned 64bit integers
GenesisAuthorities	The authority list for the genesis epoch as defined in Definition 63.	Array of tuples containing a 256-bit byte array and a unsigned 64bit integer
Randomness	The randomness for the genesis epoch	32-byte array
SecondarySlot	Whether this chain should run with round-robin-style secondary slot and if this secondary slot requires the inclusion of an auxiliary VRF output (Section 5.2.2).	A one-byte enum as defined in Definition 64 as 2_{nd}.

26.8.2. BabeApi_current_epoch_start

Finds the start slot of the current epoch.

Arguments

- None.

Return

- A unsigned 64-bit integer indicating the slot number.

26.8.3. BabeApi_current_epoch

Produces information about the current epoch.

Arguments

- None.

Return

- A data structure of the following format:

$$(e_i, s_s, d, A, r)$$

  where

  - e_i is a unsigned 64-bit integer representing the epoch index.
  - s_s is a unsigned 64-bit integer representing the starting slot of the epoch.
  - d is a unsigned 64-bit integer representing the duration of the epoch.
  - A is an authority list as defined in Definition 63.
  - r is an 256-bit array containing the randomness for the epoch as defined in Definition 79.

26.8.4. BabeApi_next_epoch

Produces information about the next epoch.

Arguments

- None.

Return

- Returns the same datastructure as described in Section 26.8.3.

26.8.5. BabeApi_generate_key_ownership_proof

Generates a proof of the membership of a key owner in the specified block state. The returned value is used to report equivocations as described in Section 26.8.6.

Arguments

- The unsigned 64-bit integer indicating the slot number.
- The 256-bit public key of the authority.

Return

- A SCALE encoded *Option* as defined in Definition Definition 164 containing the proof in an opaque form.

26.8.6. BabeApi_submit_report_equivocation_unsigned_extrinsic

A BABE equivocation occurs when a validator produces more than one block at the same slot. The proof of equivocation are the given distinct headers that were signed by the validator and which include the slot number. The Polkadot Host is expected to identify equivocators and report those to the Runtime using this function.

If there are more than two blocks which cause an equivocation, the equivocation only needs to be reported once i.e. no additional equivocations must be reported for the same slot.

Arguments

- The equivocation proof of the following format:

$$B_{\text{Ep}} = (A_{\text{id}}, s, h_1, h_2)$$

  where

  - A_{id} is the public key of the equivocator.
  - s is the slot as described in [sect-babe] at which the equivocation occurred.
  - h_1 is the block header of the first block produced by the equivocator.
  - h_2 is the block header of the second block produced by the equivocator.

    Unlike during block execution, the Seal in both block headers is not removed before submission. The block headers are submitted in its full form.

- An proof of the key owner in an opaque form as described in Section 26.8.5.

Return

- A SCALE encoded *Option* as defined in Definition 164 containing an empty value on success.

26.8.7. AuthorityDiscoveryApi Module (Version 1)

All calls in this module require (Section Section 26.1.3) to be called beforehand.

26.8.7.1. AuthorityDiscoveryApi_authorities

A function which helps to discover authorities.

Arguments

- None.

Return

- A byte array of varying size containing 256-bit pulic keys of the authorities.

26.8.8. SessionKeys Module (Version 1)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.8.8.1. `SessionKeys_generate_session_keys`

Generates a set of session keys with an optional seed. The keys should be stored within the keystore exposed by the Host Api. The seed needs to be valid and UTF-8 encoded.

Arguments

- A SCALE-encoded *Option* as defined in Definition 164 containing an array of varying sizes indicating the seed.

Return

- A byte array of varying size containing the encoded session keys.

26.8.8.2. `SessionKeys_decode_session_keys`

Decodes the given public session keys. Returns a list of raw public keys including their key type.

Arguments

- An array of varying size containing the encoded public session keys.

Return

- An array of varying size containing tuple pairs of the following format:

$$(k, k_{\text{id}})$$

  where k is an array of varying sizes containing the raw public key and k_{id} is a 4-byte array indicating the key type.

26.9. AccountNonceApi Module (Version 1)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.9.1. `AccountNonceApi_account_nonce`

Get the current nonce of an account. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section 24.1.

Arguments

- The 256-bit public key of the account.

Return

- A 32-bit unsigned integer indicating the nonce of the account.

26.9.2. TransactionPaymentApi Module (Version 1)

All calls in this module require `Core_initialize_block` (Section 26.1.3) to be called beforehand.

26.9.2.1. `TransactionPaymentApi_query_info`

Returns information of a given extrinsic. This function is not aware of the internals of an extrinsic, but only interprets the extrinsic as some encoded value and accounts for its weight and length, the Runtime's extrinsic base weight and the current fee multiplier.

This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section 24.1.

Arguments

- A byte array of varying sizes containing the extrinsic.
- The length of the extrinsic. [To do: why is this needed?]

Return

- A data structure of the following format:

$$(w, c, f)$$

  where

  - w is the weight of the extrinsic.
  - c is the "class" of the extrinsic, where class is a varying data (Definition 162) type defined as:

$$c = \begin{cases} 0 & \text{Normal extrinsic} \\ 1 & \text{Operational extrinsic} \\ 2 & \text{Mandatory extrinsic, which is always included} \end{cases}$$

  - f is the inclusion fee of the extrinsic. This does not include a tip or anything else that depends on the signature.

26.9.2.2. `TransactionPaymentApi_query_fee_details`

Query the detailed fee of a given extrinsic. This function can be used by the Polkadot Host implementation when it seems appropriate, such as for the JSON-RPC API as described in Section 24.1.

Arguments

- A byte array of varying sizes containing the extrinsic.
- The length of the extrinsic.

Return

- A data structure of the following format:

$$(f, t)$$

  where

- f is a SCALE encoded as defined in Definition 164 containing the following data structure:

$$f = (f_b, f_l, f_a)$$

where * f_b **is the minimum required fee for an extrinsic.** * f_l is the length fee, the amount paid for the encoded length (in bytes) of the extrinsic. * f_a is the "adjusted weight fee", which is a multiplication of the fee multiplier and the weight fee. The fee multiplier varies depending on the usage of the network.

- t is the tip for the block author.

Appendix A: Bibliography

[1] M. J. Saarinen and J.-P. Aumasson, “The BLAKE2 cryptographic hash and message authentication code (MAC),” -, https://tools.ietf.org/html/rfc7693, RFC 7693, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7693.

Glossary

P_n

A path graph or a path of n nodes.

$(b_0, b_1, ..., b_{n-1})$

A sequence of bytes or byte array of length n

Unknown characterUnknown characterUnknown characterUnknown character$_n$

A set of all byte arrays of length n

$I = (B_n$Unknown characterUnknown characterUnknown character$B_0)_{256}$

A non-negative interger in base 256

$B = (b_0, b_1,$ Unknown characterUnknown characterUnknown character$, b_n)$

The little-endian representation of a non-negative interger $I = (B_n$Unknown characterUnknown characterUnknown character$B_0)_{256}$ such that b_iUnknown characterUnknown characterUnknown characterB_i

Enc_{LE}

The little-endian encoding function.

C

A blockchain defined as a directed path graph.

Block

A node of the directed path graph (blockchain) C

Genesis Block

The unique sink of blockchain C

Head

The source of blockchain C

$P(B)$

The parent of block B

UNIX time

The number of milliseconds that have elapsed since the Unix epoch as a 64-bit integer

BT

The block tree of a blockchain

G

The genesis block, the root of the block tree BT

CHAIN(B)

The path graph from G to B in BT.

$Head(C)$

The head of chain C.

$|C|$

The length of chain C as a path graph

SubChain(B', B)

The subgraph of $Chain(B)$ path graph containing both B and B'.

Unknown characterUnknown characterUnknown character$_B(BT)$

The set of all subchains of BT rooted at block B.

Unknown characterUnknown characterUnknown character, Unknown characterUnknown characterUnknown character(BT)

Unknown characterUnknown characterUnknown character$_G(BT)$ i.e. the set of all chains of BT rooted at genesis block

Longest-Chain(BT)

The longest sub path graph of BT i.e. $C : |C| = \max_{C_i}$Unknown characterUnknown characterUnknown characterUnknown characterUnknown characterUnknown character$|C_i|$

Longest-Path(BT)

The longest sub path graph of $(P)BT$ with earliest block arrival time

Deepest-Leaf(BT)

Head(Longest-Path(BT)) i.e. the head of Longest-Path(BT)

$B>B'$

B is a descendant of B' in the block tree

StoredValue(k)

The function to retrieve the value stored under a specific key in the state storage.

State trie, trie

The Merkle radix-16 Tree which stores hashes of storage enteries.

KeyEncode(k)

The function to encode keys for labeling branaches of the trie.

Unknown characterUnknown characterUnknown characterUnknown character

The set of all nodes in the Polkadot state trie.

N

An individual node in the trie.

Unknown characterUnknown characterUnknown characterUnknown character$_b$

A branch node of the trie which has at least one and at most 16 children

Unknown characterUnknown characterUnknown characterUnknown character$_l$

A childless leaf node of the trie

pk_N^{Agr}

The aggregated prefix key of node N

pk_N

The (suffix) partial key of node N

Index$_N$

A function returning an integer in range of {0, . . . ,15} represeting the index of a child node of node N among the children of N

v_N

Node value containing the header of node N, its partial key and the digest of its childern values

Head$_N$

The node header of trie node N storing information about the node's type and kay

$H(N)$

The Merkle value of node N.

ChildrenBitmap

The binary function indicating which child of a given node is present in the trie.

sv_N

The subvalue of a trie node N.

Child storage

A sub storage of the state storage which has the same structure although being stored separately

Child trie

State trie of a child storage

Transaction Queue

See Definition 27.

H_p

The 32-byte Blake2b hash of the header of the parent of the block.

H_i, $H_i(B)$

Block number, the incremental interger index of the current block in the chain.

H_r

The hash of the root of the Merkle trie of the state storage at a given block

H_e

An auxileray field in block header used by Runtime to validate the integrity of the extrinsics composing the block body.

H_d, $H_d(B)$

A block header used to store any chain-specific auxiliary data.

$H_h(B)$

The hash of the header of block B

$\text{Body}(B)$

The body of block B consisting of a set of extrinsics

$M_v^{r,\ stage}$

Vote message broadcasted by the voter v as part of the finality protocol

$M_v^{r,\ Fin}(B)$

The commit message broadcasted by voter v indicating that they have finalized bock B in round r

v

GRANDPA voter node which casts vote in the finality protocol

k_v^{pr}

The private key of voter v

v_{id}

The public key of voter v

Unknown characterUnknown characterUnknown characterUnknown character$_B$, Unknown characterUnknown characterUnknown characterUnknown character

The set of all GRANDPA voters for at block B

GS

GRANDPA protocol state consisting of the set of voters, number of times voters set has changed and the current round number.

r

The voting round counter in the finality protocol

$V_(B)$

A GRANDPA vote casted in favor of block B

$V_v^{r,\ pv}$

A GRANDPA vote casted by voter v during the pre-vote stage of round r

$V_v^{r,\ pc}$

A GRANDPA vote casted by voter v during the pre-commit stage of round r

$J^{r,\ stage}(B)$

The justification for pre-committing or committing to block B in round r of finality protocol

$Sign_{v_i}^{r,\ stage}(B)$

The signature of voter v on their voteto block B, broadcasted during the specified stage of finality round r

Unknown characterUnknown characterUnknown character$^{r,\ stage}$

The set of all equivocator voters in sub-round “stage" of round r

Unknown characterUnknown characterUnknown character$_{obs(v)}^{r,\ stage}$

The set of all equivocator voters in sub-round “stage" of round r observed by voter v

$VD_{obs(v)}^{r,\ stage}(B)$

The set of observed direct votes for block B in round r

$V_{obs(v)}^{r,\ stage}$

The set of total votes observed by voter v in sub-round “stage" of round r

$V_{obs(v)}^{r,\ stage}(B)$

The set of all observed votes by v in the sub-round “stage” of round r (directly or indirectly) for block B

$B_v^{r,\ pv}$

The currently pre-voted block in round r. The GRANDPA GHOST of round r

Account key, $(sk^a,\ pk^a)$

A key pair of types accepted by the Polkadot protocol which can be used to sign transactions

$Enc_{SC}(A)$

SCALE encoding of value A

TUnknown characterUnknown characterUnknown character$(A_1,\ ...,\ A_n)$

A tuple of values A_i's each of different type

Varying Data Types

Unknown characterUnknown characterUnknown characterUnknown character = T_1, Unknown characterUnknown characterUnknown character, T_n

A data type representing any of varying types T_1, Unknown characterUnknown characterUnknown character, T_n.

SUnknown characterUnknown characterUnknown characterA_1, Unknown characterUnknown characterUnknown character, A_n

Sequence of values A_i of the same type

$Enc_{SC}^{Len}(n)$

SCALE length encoding aka. compact encoding of non-negative interger n of arbitrary size.

$Enc_{HE}(PK)$

Hex encoding