Polkadot Protocol Spec

Table of contents:

Polkadot Protocol

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

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 Polkadot Runtime and the Polkadot 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.

Polkadot Host

With the current document, we aim to specify the Polkadot Host part of the Polkadot protocol as a replicated state machine. After defining the different types of hosts in Chapter 1, we proceed to specify the representation of a valid state of the Protocol in Chapter 2. We also 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, in the same chapter. 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 and Chapter 6, we specify the consensus protocol, which is responsible for keeping all the replicas in the same state. Finally, the initial state of the machine is identified and discussed in Section A.3.3.. A Polkadot Host implementation that conforms with this part of the specification should successfully be able to sync its states with the Polkadot network.

📄 8. Availability & Validity

Polkadot serves as a replicated shared-state machine designed to resolve scalability issues and interoperability among blockchains. The validators of Polkadot execute transact…

1. Overview

The Polkadot Protocol differentiates between different classes of Polkadot Hosts. Each class differs in its trust roots and how active or passively they interact with the network.

1.1. Light Client

The light client is a mostly passive participant in the protocol. Light clients are designed to work in resource-constrained environments like browsers, mobile devices, or even on-chain. Its main objective is to follow the chain, make queries to the full node on specific information on the recent state of the blockchain, and add extrinsics (transactions). It does not maintain the full state, but rather queries the full node on the latest finalized state and verifies the authenticity of the responses trustlessly. Details of specifications focused on Light Clients can be found in Chapter 7.

1.2. Full Node

While the full node is still a mostly passive participant of the protocol, they follow the chain by receiving and verifying every block in the chain. It maintains a full state of the blockchain by executing the extrinsics in blocks. Their role in the consesus mechanism is limited to following the chain and not producing the blocks.

- **Functional Requirements:**
    i. The node must populate the state storage with the official genesis state, elaborated further in Section A.3.3..
    ii. 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.)
    iii. The node should open and maintain the various required streams (Section 4.7.) with each of its active peers.
    iv. 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.
    v. The node should exchange neighbor packets (Section 4.8.6.1.).

1.3. Authoring Node

The authoring node covers all the features of the full node, but instead of just passively following the protocol, it is an active participant, producing blocks and voting in Grandpa.

- **Functional Requirements:**
    i. Verify that the Host's session key is included in the current Epoch's authority set (Section 3.3.1.).
    ii. Run the BABE lottery (Chapter 5) and wait for the next assigned slot in order to produce a block.
    iii. Gossip any produced blocks to all connected peers (Section 4.8.1.).
    iv. Run the catch-up protocol (Section 6.6.1.) to make sure that the node is participating in the current round and not a past round.
    v. Run the GRANDPA rounds protocol (Chapter 6).

1.4. Relaying Node

The relaying node covers all the features of the authoring node but also participants in the availability and validity process to process new parachain blocks as described in Chapter 8.

2. States and Transitions

2.1. Introduction

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 inputs.
- 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 $(v_1, \ldots, v_n)$ 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 10), making the path graph tamper-proof since any modifications to the child would result in its hash value being changed.

> **INFO**
> 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.

2.1.1. 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 4. 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 (Chapter 6).

Definition 5. 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 85). By **pruning**, we refer to the procedure of $BT \leftarrow \mathrm{PBT}$. When there is no risk of ambiguity and it is safe to prune BT, we use BT to refer to PBT.

Definition 6 gives the means to highlight various branches of the block tree.

Definition 6. Subchain

Let G be the root of the block tree and B be one of its nodes. By $\mathrm{Chain}(B)$, we refer to the path graph from G to B in BT. Conversely, for a chain $C = \mathrm{Chain}(B)$, 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 $\mathrm{Chain}(B)$, then by $\mathrm{SubChain}(B', B)$ we refer to the subgraph of $\mathrm{Chain}(B)$ path graph which contains B and ends at B' and by $|\mathrm{SubChain}(B', B)|$ 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 7. 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 63) 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 $\mathrm{Longest\text{-}Chain}(BT)$ to be the maximum chain given by this order.

Definition 8. Longest Path

$\mathrm{Longest\text{-}Path}(BT)$ returns the path graph of BT which is the longest among all paths in BT and has the earliest block arrival time (Definition 63). $\mathrm{Deepest\text{-}Leaf}(BT)$ returns the head of $\mathrm{Longest\text{-}Path}(BT)$ 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 9. 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 $(|B| < |B'|) \in C$.

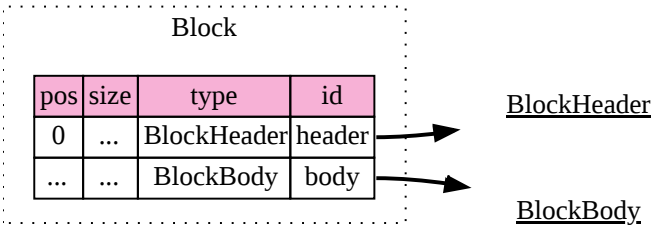
2.2. State Replication

Polkadot nodes replicate each other's states by syncing the histories of the extrinsics. This, however, is only practical if a large set of transactions are batched and synced at the same time. The structure in which the transactions are journaled and propagated is known as a block of extrinsics (Section 2.2.1.). Like any other replicated state machine, state inconsistencies can occur between Polkadot replicas. Section 2.4.5. gives an overview of how a Polkadot Host node manages multiple variants of the state.

2.2.1. Block Format

A Polkadot block consists a *block header* (Definition 10) and a *block body* (Definition 13). 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.

Image 1. Block

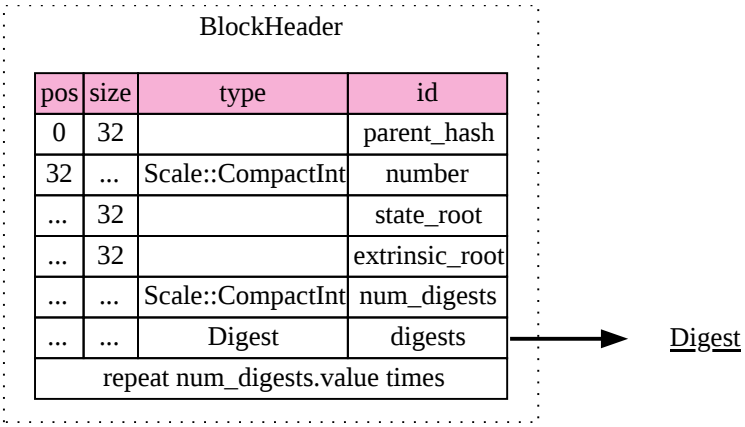


Definition 10. Block Header

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 A.1.1.1.) of the SCALE encoded parent block header (Definition 12).
- **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 the 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 11).

Image 2. Block Header



Definition 11. 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 178) such that:

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

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

$$H_d^i = \begin{cases} 4 \ \rightarrow \ (t, \mathrm{id}, m) \\ 5 \ \rightarrow \ (t, \mathrm{id}, m) \\ 6 \ \rightarrow \ (t, \mathrm{id}, m) \\ 8 \ \rightarrow \ (t) \end{cases}$$

where

- id is a 4-byte ASCII encoded consensus engine identifier
- m is a SCALE-encoded byte array containing the message payload

$t = 4$ **Consensus Message**, contains scale-encoded message m from the Runtime to the consensus engine. The receiving engine is determined by the *id* identifier:

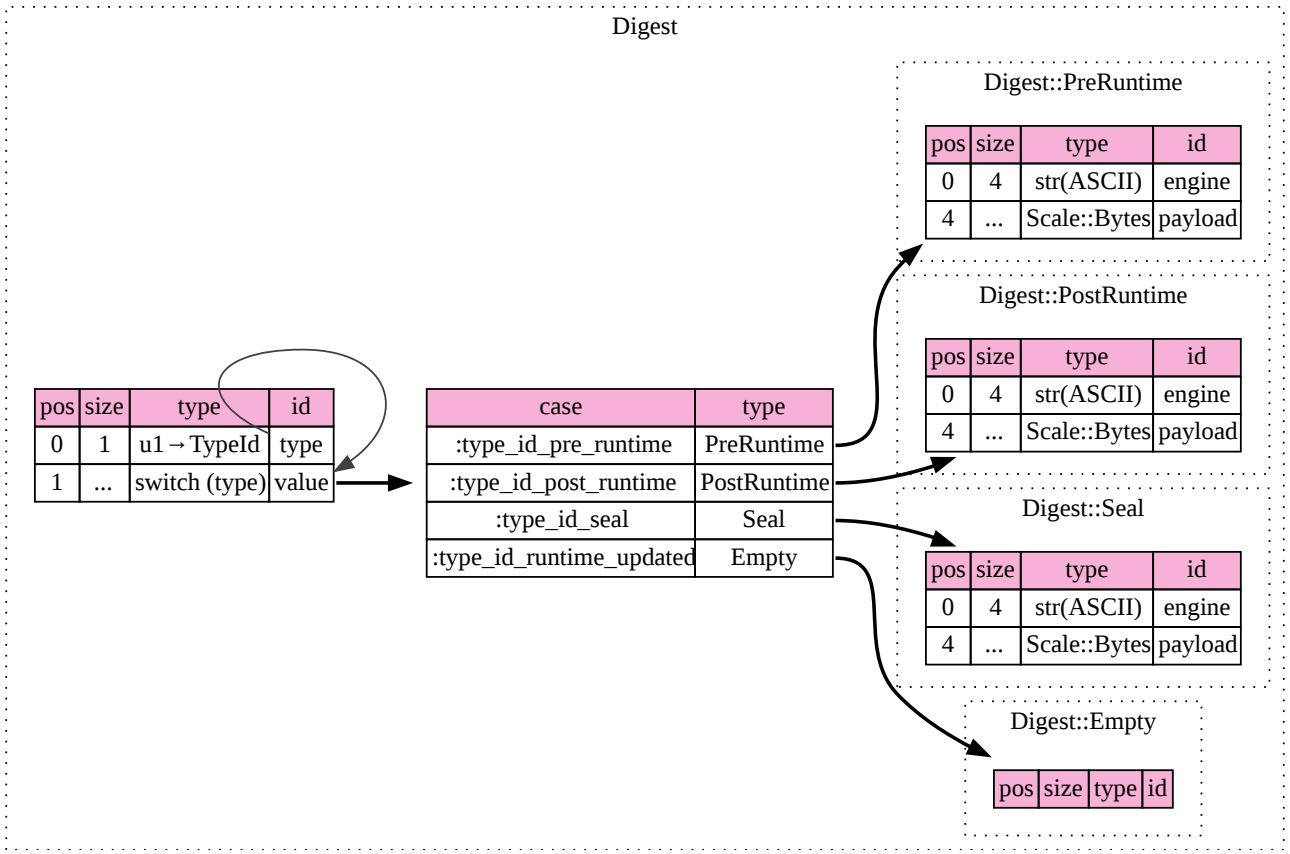
- *id* = BABE: a message to BABE engine (Definition 54)
- *id* = FRNK: a message to GRANDPA engine (Definition 82)

$t = 5$ **Seal**, is produced by the consensus engine and proves the authorship of the block producer. The engine used for this is provided through *id* (at the moment, BABE), while m contains the scale-encoded signature (Definition 66) 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.

$t = 6$ **Pre-Runtime digest**, contains messages from the consensus engines to the runtime. Currently only used by BABE to pass the scale encoded BABE Header (Definition 65) in m with *id* = BABE.

$t = 8$ **Runtime Environment Updated digest**, indicates that changes regarding the Runtime code or heap pages (Section 2.6.3.1.) occurred. No additional data is provided.

Image 3. Digest



Definition 12. 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:

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

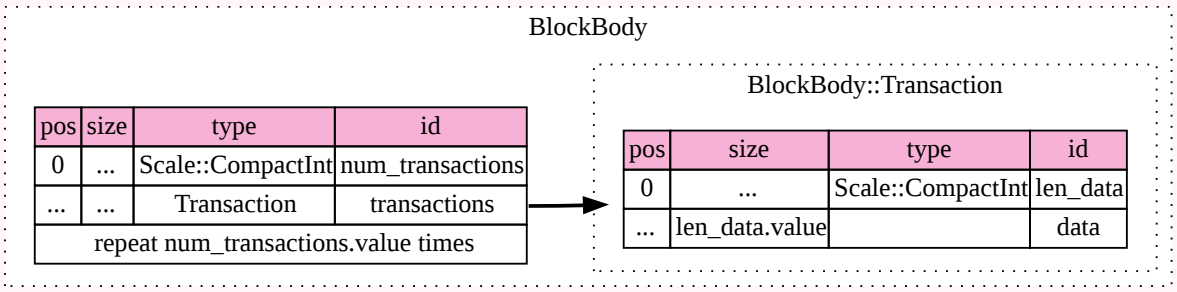
Definition 13. Block Body

The block body consists of a 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:

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

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

Image 4. Block Body



2.3. 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.

2.3.1. 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 A.1.4.) and broadcasted between the nodes. **Inherent extrinsics** are unsigned extrinsics that 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 A.2.2.).

2.3.2. 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.

Alternatively, transactions can be submitted to the host by off-chain worker through the Host API (Section B.6.2.).

Any new transaction should be submitted to the Runtime (Section C.7.1.). This will allow the Polkadot Host to check the validity of the received transaction against the current state and if it should be gossiped to other peers. If it considers the submitted transaction as valid, the Polkadot Host should store it for inclusion in future blocks. The whole process of handling new transactions is described in more detail by Validate-Transactions-and-Store.

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 14. 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, Validate-Transactions-and-Store updates the transaction pool and the transaction queue according to the received message:

Algorithm 1. Validate Transactions and Store

Algorithm Validate-Transactions-and-Store

1: $L \leftarrow Dec_{SC}(M_T)$
2: **for all** $\{T \in L \mid T \notin TQ \mid T \notin TP\}$ **do**
3: $B_d \leftarrow \text{Head}(\text{Longest-Chain}(BT))$
4: $N \leftarrow H_n(B_d)$
5: $R \leftarrow \text{Call-Runtime-Entry}(\texttt{TaggedTransactionQueue\_validate\_transaction}, N, T)$
6: **if** $\text{Valid}(R)$ **then**
7: **if** $\text{Requires}(R) \subset \bigcup_{\forall T \in (TQ \cup B_i \mid \exists i_{|d>i})} \text{Provided-Tags}(T)$ **then**
8: $\text{Insert-At}(TQ, T, \text{Requires}(R), \text{Priority}(R))$
9: **else**
10: $\text{Add-To}(TP, T)$
11: **end if**
12: $\text{Maintain-Transaction-Pool}()$
13: **if** $\text{ShouldPropagate}(R)$ **then**
14: $\text{Propagate}(T)$
15: **end if**
16: **end if**
17: **end for**

where

- M_T is the transaction message (offchain transactions?)
- Dec_{SC} decodes the SCALE encoded message.
- Longest-Chain is defined in Definition 7.
- `TaggedTransactionQueue_validate_transaction` is a Runtime entrypoint specified in Section C.7.1. and $Requires(R)$, $Priority(R)$ and $Propagate(R)$ refer to the corresponding fields in the tuple returned by the entrypoint when it deems that T is valid.
- $\text{Provided-Tags}(T)$ 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.
- $\text{Insert-At}(TQ, T, \text{Requires}(R), \text{Priority}(R))$ 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 Maintain-Transaction-Pool.
- ShouldPropagate indicates whether the transaction should be propagated based on the `Propagate` field in the `ValidTransaction` type as defined in Definition 218, which is returned by $\text{TaggedTransactionQueue\_validate\_transaction}$.
- $\text{Propagate}(T)$ sends T to all connected peers of the Polkadot Host who are not already aware of T.

Algorithm 2. Maintain Transaction Pool

Algorithm Maintain-Transaction-Pool

1: Scan the pool for ready transactions
2: Move them to the transaction queue
3: Drop invalid transactions

> **INFO**
> This has not been defined yet.

2.3.3. Inherents

Inherents are unsigned extrinsics 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 $\mathrm{BlockBuilder\_inherent\_extrinsics}$ (Section C.6.3.). Then the returned extrinsics should be included in the current block as explained in Build-Block.

Table 1. Inherent Data

Identifier	Value Type	Description
timstap0	Unsigned 64-bit integer	Unix epoch time (Definition 171)
babeslot	Unsigned 64-bit integer	The babe slot (*DEPRECATED*) (Definition 50)
parachn0	Parachain inherent data (Definition 93)	Parachain candidate inclusion (Section 8.2.2.)

Definition 15. Inherent Data

`Inherent-Data` is a hashtable (Definition 182), 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 (Build-Block).

2.4. State 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 on the size of the key or 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 A.2.2.1.).

2.4.1. Accessing System Storage

The Polkadot Host implements various functions to facilitate access to the system storage for the Runtime (Section 2.6.1.). Here we formalize the access to the storage when it is being directly accessed by the Polkadot Host (in contrast to Polkadot runtime).

Definition 16. Stored Value

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

$$\mathrm{StoredValue}\colon \mathcal{K} \to \mathcal{V}$$

$$k \to \begin{cases} v \text{ if } (k, v) \text{ exists in state storage} \\ \phi \text{ otherwise} \end{cases}$$

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.

2.4.2. General 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 *radix tree*, which hereafter we refer to as the ***State Trie*** or just ***Trie***. This rearrangement 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 2.4.4.) of the state, H_r (Definition 10), 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 17). 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 17. 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 18. Key Encode

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

$$k_{\text{enc}} = (k_{\text{enc}_1}, \dots, k_{\text{enc}_{2n}}) = \text{KeyEncode}(k)$$

such that:

$$\text{KeyEncode} : \mathbb{B} \to \text{Nibbles}^4$$

$$k \longmapsto (k_{\text{enc}_1}, \dots, k_{\text{enc}_{2n}})$$

$$(b_1, \dots, b_n) \longmapsto \left(b_1^1, b_1^2, b_2^1, b_2^2, \dots, b_n^1, b_n^2\right)$$

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:

$$k_{\text{enc}_i} = \left(b_i^1, b_i^2\right) = (b_i \div 16, b_i \text{mod} 16)$$

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.

2.4.3. Trie Structure

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

Definition 19. 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 20. State Trie

The state trie is a radix-16 tree (Definition 17). 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 21. 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 22).

Definition 22. Aggregated Prefix Key

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

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

where pk_N^{Agr} is a prefix subsequence of k_N; i is the length of pk_N^{Agr} in nibbles and so we have:

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

Part of pk_N^{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 23).

Definition 23. Index

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

$$\begin{gathered}\text{Index}_N : \{N_C \in cc(N) \mid N_c \text{ is a child of } N\} \to \text{Nibbles}_1^4 \\ N_c \to i\end{gathered}$$

such that

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

Algorithm 3. Aggregate-Key

Algorithm Aggregate-Key

Require: $P_N := (\text{TrieRoot} = N_1, \ldots, N_j = N)$
1: $pk_N^{Agr} \leftarrow \phi$
2: $i \leftarrow 1$
3: **for all** $N_i \in P_N$ **do**
4: $\quad pk_N^{Agr} \leftarrow pk_N^{Agr} || pk_{N_i} || \text{Index}_{N_i}(N_{i+1})$
5: **end for**
6: $pk_N^{Agr} \leftarrow pk_N^{Agr} || pk_N$
7: **return** pk_N^{Agr}

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

Definition 24. 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 25
- pk_N is the partial key from Definition 22
- Enc_{HE} is hex encoding (Definition 189)
- sv_N is the node subvalue from Definition 27

Definition 25. Node Header

The **node header**, consisting of ≥ 1 bytes, $N_1 \dots N_n$, specifies the node variant and the partial key length (Definition 22). 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.

$$v = \begin{cases} 01 & \text{Leaf} & p_l = 2^6 \\ 10 & \text{Branch Node with } \ k_N \notin \mathcal{K} & p_l = 2^6 \\ 11 & \text{Branch Node with } \ k_N \in \mathcal{K} & p_l = 2^6 \\ 001 & \text{Leaf containing a hashed subvalue} & p_l = 2^5 \\ 0001 & \text{Branch containing a hashed subvalue} & p_l = 2^4 \\ 00000000 & \text{Empty} & p_l = 0 \\ 00000001 & \text{Reserved for compact encoding} & \end{cases}$$

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 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 28.

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

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

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.

2.4.4. 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 2.5.1. 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 26 to encode and decode the information stored in a branch node.

Definition 26. 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_b has a child with index i, therefore we define **ChildrenBitmap** functions as follows:

$$\text{ChildrenBitmap}:$$

$$\mathcal{N}_b \to \mathbb{B}_2$$

$$N_b \to (b_{15}, \ldots, b_8, b_7, \ldots, b_0)_2$$

where

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

Definition 27. Subvalue

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

$$sv_N = \begin{cases} \text{StoredValue}_{\text{SC}} \\ \text{Enc}_{\text{SC}}(\text{ChildrenBitmap}(N) || \text{StoredValue}_{\text{SC}} || \text{Enc}_{\text{SC}}(H(N_{C_1})), \ldots, \text{Enc}_{\text{SC}}(H(N_{C_n}))) \end{cases}$$

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

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

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

- Enc_{SC} is defined in Section A.2.2..
- StoredValue, where v can be empty, is defined in Definition 16.
- H is defined in Definition 29.
- $\text{ChildrenBitmap}(N)$ is defined in Definition 26.

The trie deviates from a traditional Merkle tree in that the node value (Definition 24), v_N, is presented instead of its hash if it occupies less space than its hash.

Definition 28. 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 B.2.8.2.), then any value larger than 32 bytes is hashed, resulting in that hash being used as the subvalue (Definition 27) 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 25).

Definition 29. Merkle Value

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

$$H : \mathbb{B} \to U_{i \to 0}^{32} \mathbb{B}_{32}$$

$$H(N) : \begin{cases} v_N & ||v_N|| < 32 \text{ and } N \neq R \\ \text{Blake2b}(v_n) & ||v_N|| \geq 32 \text{ or } N = R \end{cases}$$

Where v_N is the node value of N (Definition 24) and R is the root of the trie. The **Merkle hash** of the trie is defined to be $H(R)$.

2.4.5. Managing Multiple Variants of State

Unless a node is committed to only updating its state according to the finalized block (Definition 85), 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 2.4.3.) 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 30):

Definition 30. Set State At Block

The function:

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

in which B is a block in the block tree (Definition 4), 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 2.4..

2.5. Child Storage

As clarified in Section 2.4., 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 entrypoint 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 (Section B.3.).

2.5.1. Child Tries

The child trie specification is the same as the one described in Section 2.4.3.. 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 2.4.4..

The Polkadot Host API (Section B.3.) 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.

2.6. Runtime Interactions

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 2.6.1.).

In Section 2.3., 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 a 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 2.2..

2.6.1. 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 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 Interact-With-Runtime.

Algorithm 4. Interact With Runtime

Algorithm Interact-With-Runtime

Require: $F, H_b(B), (A_1, \ldots, A_n)$
1: $\mathcal{S}_B \leftarrow$ Set-State-At$(H_b(B))$
2: $A \leftarrow Enc_{SC}((A_1, \ldots, A_n))$
3: Call-Runtime-Entrypoint$(R_B, \mathcal{RE}_B, F, A, A_{len})$

where

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

In this section, we describe the details upon which the Polkadot Host is interacting with the Runtime. In particular, Set-State-At and $\text{Call-Runtime-Entrypoint}$ procedures called by Interact-With-Runtime are explained in Definition 32 and Definition 30 respectively. R_B is the Runtime code loaded from S_B, as described in Definition 31, and RE_B is the Polkadot Host API, as described in Definition 194.

2.6.2. 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:

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

which is the ASCII byte representation of the string `:code` (Section A.3.3.). 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 point has been called. Accordingly, we define R_B (Definition 31).

The initial Runtime code of the chain is provided as part of the genesis state (Section A.3.3.) 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 (Section B.2.). Therefore, the executor **must always** load the latest Runtime from storage - or preferably detect Runtime upgrades (Definition 11) - either based on the parent block when importing blocks or the best/highest block when creating new blocks.

Definition 31. 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.

The WASM blobs may be compressed using *zstd*. In such cases, there is an 8-byte magic identifier at the head of the blob, indicating that it should be decompressed with *zstd* compression. The magic identifier prefix `ZSTD_PREFIX = [82, 188, 83, 118, 70, 219, 142, 5]` is different from the WASM magic bytes. The decompression has to be applied on the blob excluding the `ZSTD-PREFIX` and has a Bomb Limit of `CODE_BLOB_BOMB_LIMIT = 50 * 1024 * 1024` to mitigate compression bomb attacks.

2.6.3. Code Executor

The Polkadot Host executes the calls of Runtime entrypoints 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 32 introduces the notation for calling the runtime entrypoint 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 Appendix C we specify the parameters and return values for each Runtime entrypoint separately.

Definition 32. Call Runtime Entrypoint

By

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

we refer to the task using the executor to invoke the while passing an $A_1, \ldots, A_n$ argument to it and using the encoding described in Section 2.6.3.2..

2.6.3.1. 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 (Section B.10.) 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 a size of 64KB. This memory should be made available to the Polkadot Runtime for import under the symbol name `memory`.

2.6.3.2. Sending Data to a Runtime Entrypoint

In general, all data exchanged between the Polkadot Host and the Runtime is encoded using the SCALE codec described in Section A.2.2.. Therefore all runtime entrypoints have the following identical Wasm function signatures:

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

In each invocation of a Runtime entrypoints, the argument(s) which are supposed to be sent to the entrypoint, need to be SCALE encoded into a byte array B (Section A.2.2.) and copied into a section of Wasm shared memory managed by the shared allocator described in Section 2.6.3.1..

When the Wasm method, corresponding to the entrypoint, 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.

2.6.3.3. Receiving Data from a Runtime Entrypoint

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 entrypoint encoded in SCALE codec in the memory buffer. The most significant one provides the size of the blob.

2.6.3.4. Runtime Version Custom Section

For newer Runtimes, the Runtime version (Section C.4.1.) can be read directly from the Wasm custom section with the name `runtime_version.` The content is a SCALE encoded structure as described in Section C.4.1..

Retrieving the Runtime version this way is preferred over calling the `Core_version` entrypoint since it involves significantly less overhead.

3. Synchronization

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 interested 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 implementations to quickly retrieve the required information, different types of synchronization protocols are available, respectively Full, Fast, and Warp sync suited for different needs.

The associated network messages are specified in Section 4.8..

3.1. 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 45). This protocol allows nodes to arrive at the desired state much faster than fast sync.

3.2. Fast Sync

Fast sync works by downloading the block header history and validating the authority set changes (Section 3.3.1.) 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.

3.3. Full Sync

The full sync protocol 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 implementation only wants the latest, finalized state, it can combine it with protocols such as fast sync (Section 3.2.) and/or warp sync (Section 3.1.) to make synchronization as fast as possible.

3.3.1. 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 33) for that block.

Definition 33. Authority List

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

$$(pk_A, w_A)$$

pk_A is the session public key (Definition 170) of authority A. And w_A is an unsigned 64-bit integer indicating the authority weight. The value of $\mathrm{Auth}_C(B)$ is part of the Polkadot state. The value for $\mathrm{Auth}_C(B_0)$ is set in the genesis state (Section A.3.3.) and can be retrieved using a runtime entrypoint corresponding to consensus engine C.

The authorities and their corresponding weights can be retrieved from the Runtime (Section C.10.1.).

INFO

In Polkadot, the authorities are unweighted, i.e., the weights for all authorities are set to 1. The proportionality in terms of stakes is managed by the NPOS (Nominated Proof-of-Stake) algorithm in Polkadot. Once validators are elected for an era using the NPOS algorithm, they are considered equal in the BABE and GRANDPA consensus algorithms.

3.3.2. Runtime-to-Consensus Engine Message

The authority list (Definition 33) 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 in the form of a digest item (Definition 11) 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 they came into effect would get rejected.

3.4. 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 (Chapter 5). 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 produced 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 C.4.2.) as a part of the validation process. Any state changes created by this function on successful execution are persisted.

The Polkadot Host implements Import-and-Validate-Block to assure the validity of the block.

Algorithm 5. Import-and-Validate-Block

Algorithm Import-and-Validate-Block

Require: $B, \text{Just}(B)$
1: Set-Storage-State-At($P(B)$)
2: **if** $\text{Just}(B) \neq \emptyset$ **then**
3: Verify-Block-Justification($B, \text{Just}(B)$)
4: **if** B **is** Finalized **and** $P(B)$ **is not** Finalized **then**
5: Mark-as-Final($P(B)$)
6: **end if**
7: **end if**
8: **if** $H_p(B) \notin PBT$ **then**
9: **return**
10: **end if**
11: Verify-Authorship-Right(Head(B))
12: $B \leftarrow$ Remove-Seal(B)
13: $R \leftarrow$ Call-Runtime-Entry(`Core_execute_block`, B)
14: $B \leftarrow$ Add-Seal(B)
15: **if** $R =$ True **then**
16: Persist-State()
17: **end if**

where

- Remove-Seal removes the Seal digest from the block (Definition 11) before submitting it to the Runtime.
- Add-Seal adds the Seal digest back to the block (Definition 11) for later propagation.
- Persist-State implies the persistence of any state changes created by Core_execute_block (Section C.4.2.) on successful execution.
- PBT is the pruned block tree (Definition 4).
- Verify-Authorship-Right is part of the block production consensus protocol and is described in Verify-Authorship-Right.
- *Finalized block* and *finality* are defined in Chapter 6.

4. Networking

> **INFO**
> 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.
- yamux - *yamux* is a multiplexing protocol developed by HashiCorp. It is the de-facto standard for the Polkadot Host. 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. If two or more nodes use the same key, the network will interpret those nodes as a single node, which will result in unspecified behavior. 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 based on the libp2p specification, but does not fully conform to the specification. Specifically, it does not support CID and the only supported key type is ED25519.

The byte representation of the PeerId is always of the following bytes in this exact order:

$$b_0 = 0$$

$$b_1 = 36$$

$$b_2 = 8$$

$$b_3 = 1$$

$$b_4 = 18$$

$$b_5 = 32$$

$$b_{6..37} = \ldots$$

where

- b_0 is the multihash prefix of value 0 (implying no hashing is used).
- b_1 the length of the PeerId (remaining bytes).
- b_2 and b_3 are a protobuf encoded field-value pair indicating the used key type (field 1 of value 1 implies *ED25519*).
- b_4, b_5 and $b_{6..37}$ are a protobuf encoded field-value pair where b_5 indicates the length of the public key followed by the the raw ED25519 public key itself, which varies for each Polkadot Host and is always 32 bytes (field 2 contains the public key, which has a field value length prefix).

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 (Section A.3.3.).
- **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 *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/30333` to establish a TCP connection and negotiate encryption and a multiplexing layer.

- **WebSocket** with addresses in the form of `/ip4/1.2.3.4/tcp/30333/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/30333` and `/dns/example.com/tcp/30333/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 *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..

> **INFO**
> 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/1.0.0` - 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*, but the initiator does **not** send any message to the responder and only waits for the response.

  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.

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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 7.4..

> **INFO**

For backwards compatibility reasons, `/dot/light/2` is also a valid substream for those messages.

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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..

INFO

For backwards compatibility reasons, `/dot/block-announces/1` is also a valid substream for those messages.

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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..

INFO

For backwards compatibility reasons, `/dot/sync/2` is also a valid substream for those messages.

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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 4.8.4..

INFO

For backwards compatibility reasons, `/dot/sync/warp` is also a valid substream for those messages.

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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..

INFO

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

- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/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..

INFO

For backwards compatibility reasons, `/paritytech/grandpa/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.

INFO

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

$$R = \begin{cases} 1 & \text{The node is a full node} \\ 2 & \text{The node is a light client} \\ 4 & \text{The node is a validator} \end{cases}$$

$$N_B = \text{Best block number according to the node}$$

$$h_B = \text{Best block hash according to the node}$$

$$h_G = \text{Genesis block hash according to the node}$$

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

$$\text{Head}(B) = \text{Header of the announced block}$$

$$b = \begin{cases} 0 & \text{Is not part of the best chain} \\ 1 & \text{Is the best block according to the node} \end{cases}$$

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
Direction	5	Sequence direction, interpreted as Id *0* (ascending) if missing.	
`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 (enum) of the following values:

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

- *Direction* is a Protobuf structure indicating the sequence direction of the requested blocks. The structure is a varying data type (enum) 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 12
`bytes`	2	Block header (optional)	Definition 10
repeated `bytes`	3	Block body (optional)	Definition 13
`bytes`	4	Block receipt (optional)	
`bytes`	5	Block message queue (optional)	
`bytes`	6	Justification (optional)	Definition 74
`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 merkle root to confirm that claim.

See the the synchronization chapter for more information (Chapter 3).

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 chapter for more information (Chapter 3).

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 3.3.2.). It's a datastructure of the following format:

$$P = (f_x \ldots f_y, c)$$

$f_x \ldots f_y$ 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 11) 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 74) and c is a boolean that indicates whether the warp sync has been completed.

4.8.5. Transactions

Transactions (Section 2.3.) 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 2.3..

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:

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

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 Chapter 6. The underlying messages are specified in this section.

Definition 43. Grandpa Gossip Message

A **GRANDPA gossip message**, M, is a varying datatype (Definition 178) 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 72).
- $\text{id}_{\mathbb{V}}$ is an unsigned 64-bit integer indicating the authority Set Id (Definition 69).
- $\text{Sig}_{v_i}^{r,\text{stage}}$ is a 512-bit byte array containing the signature of the authority (Definition 73).
- 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 72).

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:

$$J_{v_{0,\ldots n}}^{r,\text{comp}} = \left(\left\{V_{v_0}^{r,pc}, \ldots V_{v_n}^{r,pc}\right\}, \left\{\left(\text{Sig}_{v_0}^{r,pc}, v_{\text{id}_0}\right), \ldots \left(\text{Sig}_{v_n}^{r,pc}, v_{\text{id}_n}\right)\right\}\right)$$

where

- $V_{v_i}^{r,pc}$ is a 256-bit byte array containing the pre-commit vote of authority v_i (Definition 72).
- $\text{Sig}_{v_i}^{r,pc}$ is a 512-bit byte array containing the pre-commit signature of authority v_i (Definition 73).
- 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, $M_v^{r,\text{Fin}}(B)$, is a message broadcasted by voter v to the network indicating that voter v has finalized block B in round r. It has the following structure:

$$M_v^{r,\text{Fin}}(B) = \text{Enc}_{\text{SC}}\left(r, \text{id}_{\mathbb{V}}, V_v^r(B), J_{v_{0,\ldots n}}^{r,\text{comp}}\right)$$

where

- r is an unsigned 64-bit integer indicating the round number (Definition 72).
- $id_{\mathbb{V}}$ is the authority set Id (Definition 69).
- $V_v^r(B)$ is a 256-bit array containing the GRANDPA vote for block B (Definition 71).
- $J_{v_{0,\ldots n}}^{r,\text{comp}}$ 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:

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

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 72).
- $\text{id}_{\mathbb{V}}$ is an unsigned 64-bit integer indicating the authority Id (Definition 69).
- $H_i(B_{\text{last}})$ 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 6.6.1.).

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

$$M_{v,i}^{\text{Cat}-s} = \text{Enc}_{\text{SC}}\big(\text{id}_{\mathbb{V}}, r, J_{0,\ldots n}^{r,\text{pv}}(B), J_{0,\ldots m}^{r,\text{pc}}(B), H_h(B'), H_i(B')\big)$$

Where B is the highest block which v believes to be finalized in round r (Definition 72). B' is the highest ancestor of all blocks voted on in the arrays of justifications $J_{0,\ldots n}^{r,\text{pv}}(B)$ and $J_{0,\ldots m}^{r,\text{pc}}(B)$ (Definition 74) with the exception of the equivocatory votes.

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

5. Block Production

5.1. Introduction

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 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 Block-Production-Lottery 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 recently finalized blocks (Definition 5).

5.1.1. 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.

5.1.2. Block Authoring Session Key Pair

Block authoring session key pair $\left(sk_j^s, pk_j^s\right)$ is an SR25519 key pair which the block producer $\mathcal{P}_j$ signs by their account key (Definition 167) and is used to sign the produced block as well as to compute its lottery values in Block-Production-Lottery.

Definition 50. 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.

> **INFO**
> Substrate refers to an epoch as a "session" in some places. However, epoch should be the preferred and official name for these periods. |

Definition 51. 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 C.11.1.) 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 52 provides an iterator over the blocks produced during a specific epoch.

Definition 52. Epoch Subchain

By $\mathrm{SubChain}(\mathcal{E}_n)$ 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 $\mathrm{Longest\text{-}Chain}(BT)$.

Definition 53. Equivocation

A block producer **equivocates** if they produce more than one block at the same slot. The proof of equivocation is 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 C.11.6..

Definition 54. BABE Consensus Message

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

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

where

1	implies *next epoch data*: The Runtime issues this message on every first block of an epoch. The supplied authority set Definition 33, Auth_C, and randomness Definition 67, r, are used in the next epoch $\mathcal{E}_n + 1$. In case the epochs $\mathcal{E}_n + 1$ to $\mathcal{E}_n + k$ are skipped (i.e., BABE does not produce blocks), then the epoch data (Auth_C, r) is used by the epoch $\mathcal{E}_n + k + 1$.
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's initial intention was to cause an immediate 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:

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

  where c is the probability that a slot will not be empty Definition 55. It is encoded as a tuple of two unsigned 64-bit integers $c_{nominator}, c_{denominator}$ which are used to compute the rational $c = \frac{c_{nominator}}{c_{denominator}}$.
- 2_{nd} describes what secondary slot Definition 57, if any, is to be used. It is encoded as one-byte varying datatype:

  $$s_{2\mathrm{nd}} = \begin{cases} 0 \rightarrow \text{no secondary slot} \\ 1 \rightarrow \text{plain secondary slot} \\ 2 \rightarrow \text{secondary slot with VRF output} \end{cases}$$

5.2. Block Production Lottery

The babe constant (Definition 55) is initialized at genesis to the value returned by calling `BabeApi_configuration` (Section C.11.1.). For efficiency reasons, it is generally updated by the Runtime through the *next config data* consensus message in the digest (Definition 11) 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 wins the (primary) block production lottery. Unlike the lottery, secondary slot assignees are known upfront publically (Definition 57). The Runtime provides information on how or if secondary slots are executed (Section C.11.1.), explained further in Definition 57.

Definition 55. BABE Constant

The **BABE constant** is the probability that a slot will not be empty and used in the winning threshold calculation (Definition 56). It's expressed as a rational, (x, y), where x is the numerator and y is the denominator.

Definition 56. Winning Threshold

The **Winning threshold** denoted by $T_{\mathcal{E}_n}$ is the threshold that is used alongside the result of Block-Production-Lottery 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 33) for epoch $\mathcal{E}_n$, w_a is the weight of the block author and $c \in (0, 1)$ is the BABE constant (Definition 55).

The numbers should be treated as 64-bit rational numbers.

5.2.1. Primary Block Production Lottery

A block producer aiming to produce a block during $\mathcal{E}_n$ should run the $\text{Block-Production-Lottery}$ 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

Algorithm Block-Production-Lottery

Require: sk

1: $r \leftarrow$ Epoch-Randomness(n)
2: **for** $i := 1$ **to** sc_n **do**
3: $\quad (\pi, d) \leftarrow \text{VRF}(r, i, sk)$
4: $\quad A[i] \leftarrow (d, \pi)$
5: **end for**
6: **return** A

where Epoch-Randomness is defined in (Definition 67), sc_n is defined in Definition 51 , VRF creates the BABE VRF transcript (Definition 58) and e_i is the epoch index, retrieved from the Runtime (Section C.11.1.). s_k and p_k is the secret key, respectively, the public key of the authority. For any slot s in epoch n where $o < T_{\mathcal{E}_n}$ (Definition 56), the block producer is required to produce a block.

INFO

The secondary slots (Definition 57) are running alongside the primary block production lottery and mainly serve as a fallback to in case no authority was selected in the primary lottery.

Definition 57. Secondary Slots

Secondary slots work alongside primary slot to ensure consistent block production, as described in Section 5.2.. The secondary assignee of a block is determined by calculating a specific value, i_d, which indicates the index in the authority set (Definition 33). 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 51).

$$p \leftarrow h(\text{Enc}_{\text{SC}}(r, s))$$

$$i_d \leftarrow p \bmod A_l$$

where

- r is the Epoch randomness (Definition 67).
- s is the slot number (Definition 50).
- $\text{Enc}_{\text{SC}}(\ldots)$ encodes its inner value to the corresponding SCALE value.
- $h(\ldots)$ creates a 256-bit Blake2 hash from its inner value.
- A_l is the lengths of the authority list (Definition 33).

If i_d points to the authority, that authority must claim the secondary slot by creating a BABE VRF transcript (Definition 58). The resulting values o and p are then used in the Pre-Digest item (Definition 65). In the case of secondary slots with plain outputs, respectively the Pre-Digest being of value *2*, the transcript respectively the VRF is skipped.

Definition 58. BABE Slot VRF transcript

The BABE block production lottery requires a specific transcript structure (Definition 165). That structure is used by both primary slots (Block-Production-Lottery) and secondary slots (Definition 57).

$$t_1 \leftarrow \text{Transcript}(\text{'BABE'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{'slot number'}, s)$$

$$t_3 \leftarrow \text{append}(t_2, \text{'current epoch'}, e_i)$$

$$t_4 \leftarrow \text{append}(t_3, \text{'chain randomness'}, r)$$

$$t_5 \leftarrow \text{append}(t_4, \text{'vrf-nm-pk'}, p_k)$$

$$t_6 \leftarrow \text{meta-ad}(t_5, \text{'VRFHash'}, \text{False})$$

$$t_7 \leftarrow \text{meta-ad}(t_6, 64_{\text{le}}, \text{True})$$

$$h \leftarrow \text{prf}(t_7, \text{False})$$

$$o = s_k \cdot h$$

$$p \leftarrow \text{dleq\_prove}(t_7, h)$$

The operators are defined in Definition 166, dleq_prove in Definition 162. The computed outputs, o and p, are included in the block Pre-Digest (Definition 65).

5.3. 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 60).

DANGER

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 $s = t_{\text{unix}}/\mathcal{T}$ where $\mathcal{T}$ is defined in Definition 50 and t_{unix} is defined in Definition 171. 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 61) and s_{cq} (Definition 62). 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 55) of $c = 0.38$.

All validators are then required to run Median-Algorithm at the beginning of each sync period (Definition 64) 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 61). The target slot to which to synchronize should be the first slot in the new sync period.

Definition 59. 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 60).

Definition 60. 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

Algorithm Slot-Time

Require: s

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

where s is the slot number.

Algorithm 8. Median Algorithm

Algorithm Median-Algorithm

Require: $\mathfrak{E}, s_{sync}$

1: $T_s \leftarrow \{\}$
2: **for** B **in** $\mathfrak{E}_j$ **do**
3: $\quad t_{est}^B \leftarrow T_B + \textsc{Slot-Offset}(s_B, s_{sync}) \times \mathcal{T}$
4: $\quad T_s \leftarrow T_s \cup t_{est}^B$
5: **end for**
6: **return** $\textsc{Median}(T_s)$

where

- $\mathfrak{E}$ is the sync period used for the estimate.
- s_{sync} is the slot time to estimate.
- Slot-Offset is defined in Slot-Time.
- $\mathcal{T}$ is the slot duration defined in Definition 50.

Definition 61. Pruned Best Chain

The **pruned best chain** C^{r^k} is the longest selected chain (Definition 7) 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 62. 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 63) measured by a clock that is otherwise not adjusted by any external protocol.

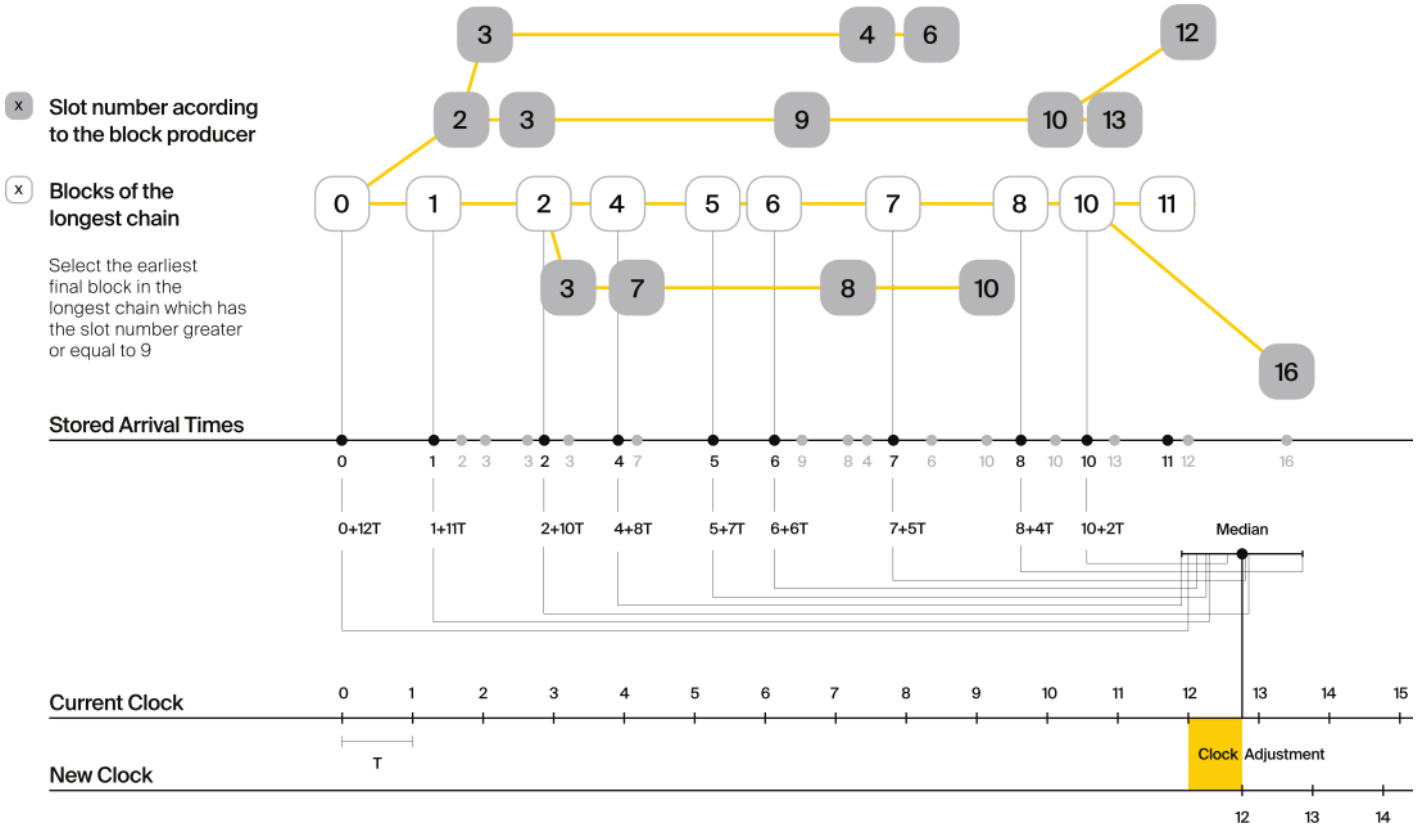
Definition 63. Block Arrival Time

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 64. 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 62) is equal to $s_{qc}$and expressed in the number of slots.

Image 5. An exemplary result of Median Algorithm in first sync epoch with $s_{\text{cq}} = 9$ and $k = 1$.



5.4. Production Algorithm

Throughout each epoch, each block producer should run Invoke-Block-Authoring to produce blocks during the slots it has been awarded during that epoch. The produced block needs to carry the *Pre-Digest* (Definition 65) as well as the *block signature* (Definition 66) as Pre-Runtime and Seal digest items.

Definition 65. Pre-Digest

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

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

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.). 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 3.3.1.) who authored the block.

- s is the slot number (Definition 50).
- o is VRF output (Block-Production-Lottery respectively Definition 57).
- p is VRF proof (Block-Production-Lottery respectively Definition 57).

The Pre-Digest must be included as a digest item of Pre-Runtime type in the header digest (Definition 11) $H_d(B)$.

Algorithm 9. Invoke-Block-Authoring

Algorithm 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 τ > d 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
```

where BT is the current block tree, $\mathrm{Block\text{-}Production\text{-}Lottery}$ is defined in Block-Production-Lottery and $\mathrm{Add\text{-}Digest\text{-}Item}$ appends a digest item to the end of the header digest $H_d(B)$ (Definition 11).

Definition 66. Block Signature

The **Block Signature** S_B is a signature of the block header hash (Definition 12) and defined as

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

m should be included in $H_d(B)$ as the Seal digest item (Definition 11) of value:

$$(t, \mathrm{id}(\mathrm{BABE}), m)$$

in which, $t = 5$ is the seal digest identifier and $\mathrm{id}(\mathrm{BABE})$ is the BABE consensus engine unique identifier (Definition 11). The Seal digest item is referred to as the **BABE Seal**.

5.5. 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 67) necessary to participate in the block production lottery in the next epoch $\mathcal{E}_{n+1}$ from the Runtime, through the consensus message (Definition 54) in the digest of the first block.

Definition 67. 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 C.11.1.). For any further epochs, the randomness is retrieved from the consensus message (Definition 54).

5.6. 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 Verify-Authorship-Right. Verify-Slot-Winner runs as part of the verification process, when a node is importing a block.

Algorithm 10. Verify Authorship Right

Algorithm Verify-Authorship-Right

Require: $\text{Head}_{s(B)}$

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

where

- $\text{Head}_s(B)$ is the header of the block that's being verified.
- T_B is B's arrival time (Definition 63).
- $H_d(B)$ is the digest sub-component (Definition 11) of $\text{Head}(B)$ (Definition 10).
- The Seal D_s is the last element in the digest array $H_d(B)$ as described in Definition 11.
- Seal-Id is the type index showing that a digest item (Definition 11) of varying type (Definition 179) is of type *Seal*.
- $\text{AuthorityDirectory}^{\mathcal{E}_c}$ is the set of Authority ID for block producers of epoch $\mathcal{E}_c$.
    i. AuthorId is the public session key of the block producer.
- BT is the pruned block tree (Definition 5).
- Verify-Slot-Winner is defined in Verify-Slot-Winner.

Algorithm 11. Verify Slot Winner

Algorithm Verify-Slot-Winner

Require: B

$\mathcal{E}_c \leftarrow$ Current-Epoch
$\rho \leftarrow$ Epoch-Randomness(c)
Verify-VRF$(\rho, H_{BABE}(B).(d_B, \pi_B), H_{BABE}(B).s, c)$
if $d_B \geqslant \tau$ **then**
    error ''Block producer is not a winner of the slot''
end if

where

1. Epoch-Randomness is defined in Definition 67.
2. $H_{\text{BABE}}(B)$ is the BABE header defined in Definition 65.
3. (o, p) is the block lottery result for block B (Block-Production-Lottery), respectively the VRF output (Definition 58).
4. Verify-VRF is described in Section A.1.3..
5. $T_{\mathcal{E}_n}$ is the winning threshold as defined in Definition 56.

5.7. Block Building Process

The block building process is triggered by Invoke-Block-Authoring of the consensus engine which in turn runs Build-Block.

Algorithm 12. Build Block

Algorithm Build-Block

1: $P_B \leftarrow$ Head(C_{Best})
2: Head$(B) \leftarrow (H_p \leftarrow H_h(P_B), H_i \leftarrow H_i(P_B) + 1, H_r \leftarrow \phi, H_e \leftarrow \phi, H_d \leftarrow \phi)$
3: Call-Runtime-Entry(`Core_initialize_block`, Head(B))
4: I-D$\leftarrow$Call-Runtime-Entry(`BlockBuilder_inherent_extrinsics`,Inherent-Data)
5: **for** E **in**I-D **do**
6: Call-Runtime-Entry(`BlockBuilder_apply_extrinsics`, E)
7: **end for**
8: **while not** End-Of-Slot(s) **do**
9: $E \leftarrow$ Next-Ready-Extrinsic()
10: $R \leftarrow$ Call-Runtime-Entry(`BlockBuilder_apply_extrinsics`, E)
11: **if** Block-Is-Full(R) **then**
12: **break**
13: **end if**
14: **if** Should-Drop(R) **then**
15: Drop(E)
16: **end if**
17: Head$(B) \leftarrow$ Call-Runtime-Entry(`BlockBuilder_finalize_block`, B)
18: $B \leftarrow$ Add-Seal(B)
19: **end while**

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 10.
- Call-Runtime-Entry is defined in Definition 32.
- Inherent-Data is defined in Definition 15.
- End-Of-Slot indicates the end of the BABE slot as defined Median-Algorithm respectively Definition 50.
- Next-Ready-Extrinsic indicates picking an extrinsic from the extrinsics queue (Definition 14).
- 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 210) describes this behavior in more detail.
- Drop indicates removing the extrinsic from the extrinsic queue (Definition 14).
- Add-Seal adds the seal to the block (<<>>) before sending it to peers. The seal is removed again before submitting it to the Runtime.

6. Finality

6.1. Introduction

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.

Definition 68. 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 entrypoint of the Runtime described in Section C.10.1.. 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 69. Authority Set Id

The **authority set Id** ($\text{id}_{\mathbb{V}}$) is an incremental counter which tracks the amount of authority list changes that occurred (Definition 82). 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 70. GRANDPA State

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

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

where

- $\mathbb{V}$: is the set of voters.
- $\text{id}_{\mathbb{V}}$: is the authority set ID (Definition 69).
- r: is the voting round number.

Definition 71. 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 12) and the block number (Definition 10).

Definition 72. 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 $V_v^{r,\text{pv}}$ and $V_v^{r,\text{pc}}$ 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 (Play-Grandpa-Round).

Definition 73. Vote Signature

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

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

where

- msg: is a byte array containing the message to be signed (Definition 71).
- r: is an unsigned 64-bit integer is the round number.
- $\text{id}_{\mathbb{V}}$: is an unsigned 64-bit integer indicating the authority set Id (Definition 69).

Definition 74. Justification

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

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

in which either

$$B' \geq B$$

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

In all cases, $\text{Sign}_{v_i}^{r,\text{stage}}(B')$ is the signature (Definition 73) of voter $v_{\text{id}} \in \mathbb{V}_B$ 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 75. Finalizing Justification

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

Note that $J^{r,\text{pc}}(B)$ 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 85) of block B by actively participating in the voting process. That is by invoking Play-Grandpa-Round.

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

Definition 76. 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 $V_v^{r,\text{stage}}(B)$ cast by v in that sub-round is an **equivocatory vote**, and

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

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:

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

The Polkadot Host must detect equivocations committed by other validators and submit those to the Runtime as described in Section C.10.3..

A vote $V^{r,\text{stage}}_{v} = V(B)$ is **invalid** if

- $H(B)$ does not correspond to a valid block.
- B is not an (eventual) descendant of a previously finalized block.
- $M^{r,\text{stage}}_{v}$ does not bear a valid signature.
- $\text{id}_{\mathbb{V}}$ does no match the current $\mathbb{V}$.
- $V^{r,\text{stage}}_{v}$ is an equivocatory vote.

Definition 77. Set of Observed Direct Votes

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

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

Definition 78. 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 79. 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:

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

Definition 80. Current Pre-Voted Block

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

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

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

Definition 81. Completable Round

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

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

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

Definition 82. GRANDPA Consensus Message

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 6.5..
3	Implies **on disabled**: An index to the individual authority in the current authority list (Definition 33) 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.

6.2. 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 Initiate-Grandpa.

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.

6.2.1. Voter Set Changes

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

> **INFO**
> 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 3.3.2.). 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 Initiate-Grandpa.

Algorithm 13. Initiate Grandpa

Algorithm 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)

where B_{last} is the last block which has been finalized on the chain (Definition 85). 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 85. 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.

6.3. 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 6.6.1.).

6.4. Voting Process in Round r

For each round r, an honest voter v must participate in the voting process by following Play-Grandpa-Round.

Algorithm 14. Play Grandpa Round

Algorithm Play-Grandpa-Round

Require: (r)

1: $t_{r,v} \leftarrow$ Current local time
2: primary $\leftarrow$ DERIVE-PRIMARY(r)
3: **if** $v =$ primary **then**
4: BROADCAST$(M_v^{r-1,\text{Fin}}$(BEST-FINAL-CANDIDATE$(r-1)))$
5: **if** BEST-FINAL-CANDIDATE$(r-1) \geqslant$ LAST-FINALIZED-BLOCK **then**
6: BROADCAST$(M_v^{r-1,\text{Prim}}$(BEST-FINAL-CANDIDATE$(r-1)))$
7: **end if**
8: **end if**
9: RECEIVE-MESSAGES(**until** Time $\geqslant t_{r,v} + 2 \times T$ **or** r **is** completable)
10: $L \leftarrow$ BEST-FINAL-CANDIDATE$(r-1)$
11: $N \leftarrow$ BEST-PREVOTE-CANDIDATE(r)
12: BROADCAST$(M_v^{r,\text{pv}}(N))$
13: RECEIVE-MESSAGES(**until** $B_v^{r,\text{pv}} \geqslant L$ **and** (Time $\geqslant t_{r,v} + 4 \times T$ **or** r **is** completable))
14: BROADCAST$(M_v^{r,\text{pc}}(B_v^{r,\text{pv}}))$

15: **repeat**
16: RECEIVE-MESSAGES()
17: ATTEMPT-TO-FINALIZE-AT-ROUND(r)
18: **until** r **is** completable **and** FINALIZABLE(r) **and** LAST-FINALIZED-BLOCK $\geqslant$ BEST-FINAL-CANDIDATE($r-1$)
19: PLAY-GRANDPA-ROUND($r+1$)
20: **repeat**
21: RECEIVE-MESSAGES()
22: ATTEMPT-TO-FINALIZE-AT-ROUND(r)
23: **until** LAST-FINALIZED-BLOCK $\geqslant$ BEST-FINAL-CANDIDATE(r)
24: **if** $r >$ LAST-COMPLETED-ROUND **then**
25: LAST-COMPLETED-ROUND $\leftarrow r$
26: **end if**

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 Derive-Primary.
- The condition of *completablitiy* is defined in Definition 81.
- Best-Final-Candidate function is explained in Best-Final-Candidate.
- Attempt-To-Finalize-At-Round(r) is described in Attempt-To-Finalize-At-Round.
- Finalizable is defined in Finalizable.

Algorithm 15. Derive Primary

Algorithm Derive-Primary

Input: r
1: **return** $r \bmod |\mathbb{V}|$

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

Algorithm 16. Best Final Candidate

Algorithm Best-Final-Candidate

Input: r
1: $B_v^{r,pv} \leftarrow$ GRANDPA-GHOST(r)
2: **if** $r = 0$ **then**
3: **return** $B_v^{r,pv}$
4: **else**
5: $\mathcal{C} \leftarrow \{B' | B' \leqslant B_v^{r,pv} | \#V^{r,pc}_{\text{obv}(v),pot}(B') > \frac{2}{3}|\mathbb{V}|\}$
6: **if** $\mathcal{C} = \phi$ **then**
7: **return** $B_v^{r,pv}$
8: **else**
9: **return** $E \in \mathcal{C} : H_n(E) = \max(H_n(B') | B' \in \mathcal{C})$
10: **end if**
11: **end if**

where $\#V^{r,pc}_{\text{obv}(v),pot}$ is defined in Definition 79.

Algorithm 17. GRANDPA GHOST

Algorithm GRANDPA-GHOST

Input: r
1: **if** $r = 0$ **then**

$G \leftarrow B_{last}$
else
$L \leftarrow$ Best-Final-Candidate$(r-1)$
$\mathcal{G} = \{\forall B > L | \#V^{r,pv}_{\text{obs}(v)}(B) \geqslant \frac{2}{3}|\mathbb{V}|\}$
if $\mathcal{G} = \phi$ **then**
$G \leftarrow L$
else
$G \in \mathcal{G} | H_n(G) = \max\left(H_n(B) | \forall B \in \mathcal{G}\right)$
end if
end if
return G

where

- B_{last} is the last block which has been finalized on the chain (Definition 85).
- $\#V^{r,pv}_{\text{obs}(v)}(B)$ is defined in Definition 78.

Algorithm 18. Best PreVote Candidate

Algorithm Best-PreVote-Candidate

Input: r
$B^{r,pv}_v \leftarrow$ GRANDPA-GHOST(r)
if Received$(M^{r,prim}_{v_{primary}}(B))$ **and** $B^{r,pv}_v \geqslant B > L)$ **then**
$N \leftarrow B$
else
$N \leftarrow B^{r,pv}_v$
end if

Algorithm 19. Attempt To Finalize At Round

Algorithm Attempt-To-Finalize-At-Round

Require: (r)
$L \leftarrow$ Last-Finalized-Block
$E \leftarrow$ Best-Final-Candidate(r)
if $E \geqslant L$ **and** $V^{r,\text{pc}}_{\text{obs}(v)}(E) > 2/3|\mathbb{V}|$ **then**
Last-Finalized-Block$\leftarrow E$
if $M^{r,\text{Fin}}_v(E) \notin$ Received-Messages **then**
Broadcast$(M^{r,\text{Fin}}_v(E))$
return
end if
end if

Algorithm 20. Finalizable

Algorithm Finalizable

Require: (r)
if r **is not** Completable **then**
return False
end if
$G \leftarrow$ GRANDPA-GHOST$(J^{r,pv}(B))$
if $G = \phi$ **then**
return False
end if
$E_r \leftarrow$ Best-Final-Candidate(r)
if $E_r \neq \phi$ **and** Best-Final-Candidate$(r-1) \leqslant E_r \leqslant G$ **then**
return True
else
return False

13: **end if**

where the condition for *completability* is defined in Definition 81.

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 Play-Grandpa-Round prevents us from moving to the round $r + 1$ before finalizing the best final candidate of round $r - 1$) The example in Definition 83 serves to demonstrate a situation where the best final candidate of a round cannot be finalized during its own round:

Definition 83. 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 $\text{GRANDPA-GHOST}(1) = B_2$.

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 $\text{Best-Final-Candidate}(r) = B_2$ 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 $\text{GRANDPA-GHOST}(1) = B_2$ 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 Play-Grandpa-Round 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 Attempt-To-Finalize-At-Round 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 $\text{Best-Final-Candidate}(r) = B_1$.

Both scenarios unblock Play-Grandpa-Round, $\text{Last-Finalized-Block} \geq \text{Best-Final-Candidate}(r - 1)$ albeit in different ways: the former with increasing the $\text{Last-Finalized-Block}$ and the latter with decreasing $\text{Best-Final-Candidate}(r - 1)$.

6.5. 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 82) 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.

Image 6. Applying a scheduled change

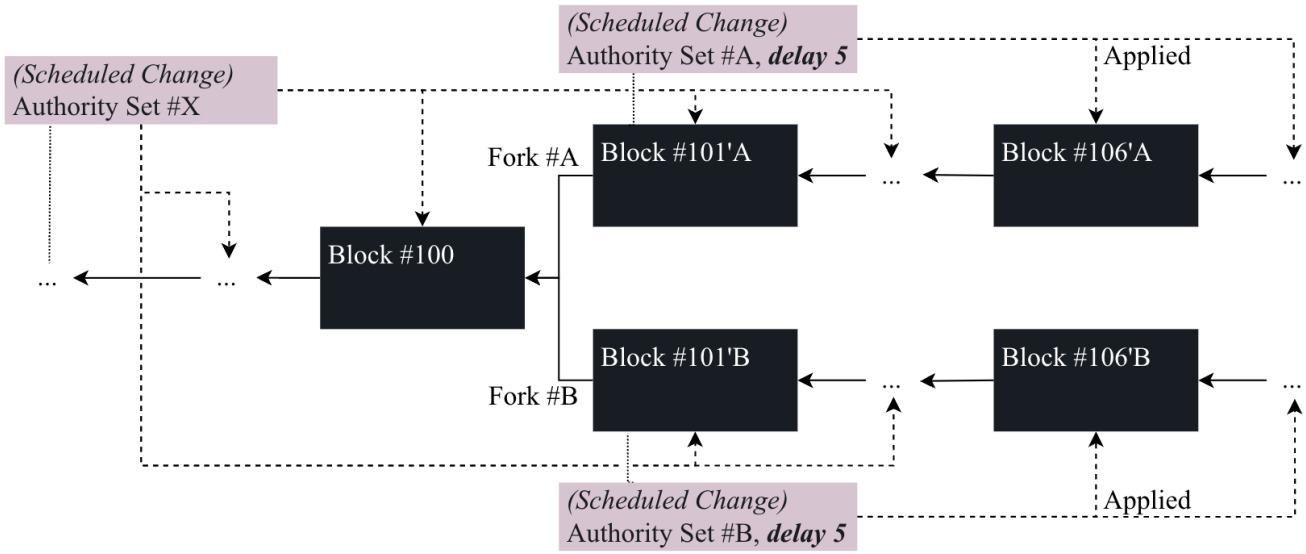
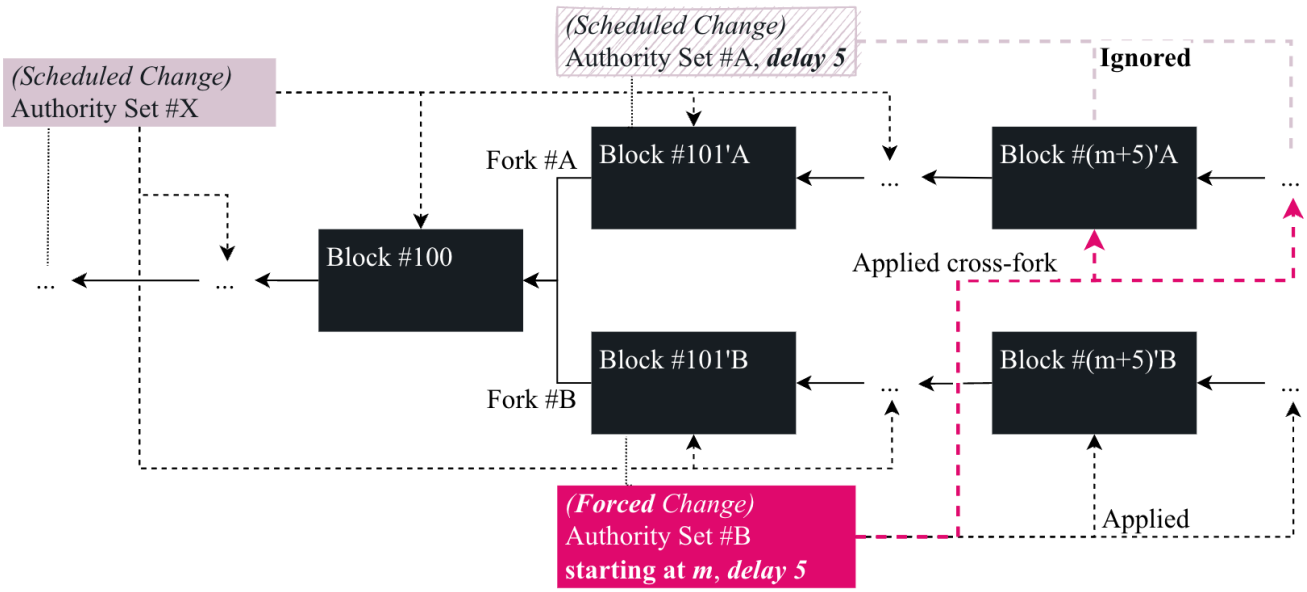


Image 7. Applying a forced change



6.6. Block Finalization

Definition 84. 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 10) and denoted by $\mathrm{Head}(B)$.
- **justification**: as defined by the consensus specification indicated by $\mathrm{Just}(B)$ as defined in Definition 74.
- **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 85. Finalized

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

- $V^{r,\text{pc}}_{\text{obs}(n)}(B') > \frac{2}{3}|\mathbb{V}_{B'}|$.
- It receives a $M^{r,\text{Fin}}_{v}(B')$ message in which $J^r(B)$ justifies the finalization (Definition 74).
- It receives a block data message for B' with $\text{Just}(B')$ (Definition 84) 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 Play-Grandpa-Round 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 6.6.1..

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

6.6.1. Catching up

When a Polkadot node (re)joins the network, 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 74) 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.

6.6.1.1. 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.

6.6.1.2. 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 Process-Catchup-Request. 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 Play-Grandpa-Round is not concluded.

Algorithm 21. Process Catchup Request

Algorithm Process-Catchup-Request

Input: $M^{\text{Cat-q}}_{i,v}(\text{id}_{\mathbb{V}}, r)$

if $M^{\text{Cat-q}}_{i,v}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}} \neq \text{id}_{\mathbb{V}}$ **then**
    error ''Catching up on different set''
end if
if $i \notin \mathbb{P}$ **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$(i, M^{\text{Cat-s}}_{v,i}(\text{id}_{\mathbb{V}}, r))$

where

- $M^{\text{Cat}-q}_{i,v}(\text{id}_{\mathbb{V}}, r)$ is the catch-up message received from peer i (Definition 48).
- $\text{id}_{\mathbb{V}}$ (Definition 69) 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 Initiate-Grandpa and gets updated by Play-Grandpa-Round.

- $M_{v,i}^{\text{Cat}-s}(\text{id}_{\mathbb{V}}, r)$ is the catch-up response (Definition 49).

6.6.1.3. 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 Process-Catchup-Response.

Algorithm 22. Process Catchup Response

Algorithm Process-Catchup-Response

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

$M_{v,i}^{\text{Cat-s}}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}}, r, J^{r,pv}(B), J^{r,pc}(B), H_h(B'), H_i(B') \leftarrow \text{Dec}_{SC}(M_{v,i}^{Cat-s}(\text{id}_{\mathbb{V}}, r)$
if $M_{v,i}^{\text{Cat-s}}(\text{id}_{\mathbb{V}}, r).\text{id}_{\mathbb{V}} \neq \text{id}_{\mathbb{V}}$ **then**
    error ''Catching up on different set''
end if
if $r \leqslant$ 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 $i \in \mathbb{V}$ **then**
    Play-Grandpa-round$(r+1)$
end if

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

7. Light Clients

7.1. Requirements for Light Clients

We list the requirements of a Light Client categorized along the three dimensions of Functionality, Efficiency, and Security.

- **Functional Requirements:**
    i. Update state (Section 2.4.) to reflect the latest view of the blockchain via synchronization with full nodes.
    ii. (Optional) Verify validity of runtime transitions (Section 2.6.).
    iii. Make queries for data at the latest block height or across a range of blocks.
    iv. Append extrinsics (Section 2.3.) to the blockchain via full nodes.
- **Efficiency Requirements:**
    i. Efficient bootstrapping and syncing: initializations and update functions of the state have tractable computation and communication complexity and grows at most linearly with the chain size. Generally, the complexity is proportional to the GRANDPA validator set change.
    ii. Querying operations happen by requesting the key-value pair from a full node.
    iii. Further, verifying the validity of responses by the full node is logarithmic in the size of the state.
- **Security Requirements:**
    i. Secure bootstrapping and Synchronizing: The probability that an adversarial full node convinces a light client of a forged blockchain state is negligible.
    ii. Secure querying: The probability that an adversary convinces a light client to accept a forged account state is negligible.
    iii. Assure that the submitted extrinsics are appended in a successor block or inform the user in case of failure.
- **Polkadot Specific Requirements:**
    i. The client MUST be able to connect to a relay chain using chain state.
    ii. The client MUST be able to retrieve the checkpoint state from a trusted source to speed up initialization.
    iii. The client MUST be able to subscribe/unsubscribe to/from any polkadot-spec-conformant relay chain (Polkadot, Westend, Kusama)
    iv. The client MUST be able to subscribe/unsubscribe to/from parachains that do not use custom protocols or cryptography methods other than those that Polkadot, Westend and Kusama use.
    v. The client MUST support the following RPC methods: `rpc_methods`, `chainHead_unstable_follow`, `chainHead_unstable_unfollow`, `chainHead_unstable_unpin`, `chainHead_unstable_storage`, `chainHead_unstable_call` `chainHead_unstable_stopCall`. `transaction_unstable_submitAndWatch`, and `transaction_unstable_unwatch`
    vi. The client MUST support the @substrate/connect connection extension protocol: `ToApplicationError`, `ToApplicationChainReady`, `ToApplicationRpc`, `ToExtensionAddChain`, `ToExtensionAddWellKnownChain`, `ToExtensionRpc`, `ToExtensionRemoveChain`.

7.2. Warp Sync for Light Clients

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 74). This protocol allows nodes to arrive at the desired state much faster than fast sync. Warp sync is primarily designed for Light Clients. Although, warp sync could be used by full nodes, the sync process may lack information to cater to complete functionality set of full nodes.

For light clients, it is too expensive to download the state (approx. 550MB) to respond to queries. Rather, the queries are submitted to the Full node, and only the response of the full node is validated using the hash of the state root. Requests for warp sync are performed using the `/dot/sync/warp` *Request-Response* substream, the corresponding network messages are detailed in Section 4.7..

Light clients base their trust in provided snapshots and the ability to slash grandpa votes for equivocation for the period they are syncing via warp sync. Full nodes and above, in contrast, verify each block individually.

In theory, the `warp sync` process takes the Genesis Block as input and outputs the hash of the state trie root at the latest finalized block. This root hash acts as proof to further validate the responses to queries by the full node. The `warp sync` works by starting from a trusted specified block (e.g., from a snapshot) and verifying the block headers only at the authority set changes.

Eventually, the light client verifies the finality of the block returned by a full node to ensure that the block is indeed the latest finalized block. This entails two things:

1. Check the authenticity of GRANDPA Justifications messages from Genesis to the last finalized block.

2. Check the timestamp of the last finalized block to ensure that no other blocks might have been finalized at a later timestamp.

> ⚠ **CAUTION**
>
> **Long-Range Attack Vulnerabilities**: Warp syncing is particularly vulnerable to what is called long-range attacks. The authorities allowed to finalize blocks can generate multiple proofs of finality for multiple different blocks of the same height. Hence, they can finalize more than one chain at a time. It is possible for two-thirds of the validators that were active at a certain past block N to collude and decide to finalize a different block N', even when N has been finalized for the first time several weeks or months in the past. When a client then warp syncs, it can be tricked to consider this alternative block N' as the finalized one. However, in practice, to mitigate Long-Range Attacks, the starting point of the warp syncing is not too far in the past. How far exactly depends on the logic of the runtime of the chain. For example, in Polkadot, the starting block for the sync should be at max 28 days old to be within the purview of the slashing period for misbehaving nodes. Hence, even though, in theory, warp sync can start from Genesis Block, it is not advised to implement the same in practice.

We outline the warp sync process, abstracting out details of verifying the finality and how the full node to sync with is selected.

Algorithm 23. Warp Sync Light Clients

Algorithm Warp-Sync-Light-Clients

Input: BlockHeader startblock, the initial block to start the sync. May not be the Genesis Block.

Output: CommitmentRootHash *root*, State Tries Root hash of the latest finalized Block.

1: fullnode ← SelectFullNode
2: latestBlockHeader, grandpaJustifications ← SyncWithNode(fullnode)
3: isVerified ← verifyAuthoritySetChange(grandpaJustifications) ∧ verifyFinality(latestBlockHeader)
4: **if** isVerified **then**
5: *return SOME* getCommitmentRootHash(latestBlockHeader)
6: **end if**
7: *throw ERROR*

Abstraction of Warp Sync and verification of the latest block's finality.

$SelectFullNode$: Determines the full node that the light client syncs with.

$SyncSithNode$: Returns the header of the latest finalized block and a list of Grandpa Justifications by the full node.

$verifyAuthoritySetChange$: Verification algorithm which checks the authenticity of the header only at the end of an era where the authority set changes iteratively until reaching the latest era.

$verifyFinalty$: Verifies the finality of the latest block using the Grandpa Justifications messages.

The warp syncing process is closely coupled with the state querying procedure used by the light client. We outline the process of querying the state by a light client and validating the response.

Algorithm 24. Querying State Light Clients

Algorithm Querying-State-Light-Clients

Input: Query q, BlockHeight h, CommitmentRootHash *root*

Output: Maybe Result *res*

1: $(res, \pi) \leftarrow QueryFullNode(q, h)$
2: **if** $validityCheck_{root}(res, \pi)$ **then**

3: $return\ SOME\ res$
4: **end if**
5: $throw\ ERROR$

Querying State Algorithm.

$QueryFullNode$: Returns the response to the query requested from the Full Node for the query q at block height h.

$validityCheck_{root}$: Predicate that checks the validity of response res and associated merkle proof π by matching it against the Commit Root Hash $root$ obtained as a result of warp sync.

7.3. Runtime Environment for Light Clients

Technically, though a runtime execution environment is not necessary to build a light client, most clients require interacting with the Runtime and the state of the blockchain for integrity checks at the minimum. One can imagine an application scenario like an on-chain light client which only listens to the latest state without ever adding extrinsics. Current implementations of Light Nodes (for e.g., Smoldot) use the wasmtime as its runtime environment to drastically simplify the code. The performance of wasmtime is satisfying enough not to require a native runtime. The details of the runtime API that the environment needs to support can be found in (Appendix C).

7.4. 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.

7.4.1. Request

A message with all possible request messages. All messages 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 86)
`RemoteReadRequest`	2	A remote read request (Definition 88)
`RemoteReadChildRequest`	4	A remote read child request (Definition 90)

7.4.2. Response

A message with all possible response messages. All messages 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 87)

Type	Id	Description
`RemoteReadResponse`	2	A remote read response (Definition 89)

7.4.3. Remote Call Messages

Execute a call to a contract at the given block.

Definition 86. 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 87. Remote Call Response

Remote call response.

Type	Id	Description
`bytes`	2	An *Option* type (Definition 180) containing the call proof or *None* if proof generation failed.

7.4.4. Remote Read Messages

Read a storage value at the given block.

Definition 88. Remote Read Request

Remote read request.

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

Definition 89. Remote Read Response

Remote read response.

Type	Id	Description
`bytes`	2	An *Option* type (Definition 180) containing the read proof or *None* if proof generation failed.

7.4.5. Remote Read Child Messages

Read a child storage value at the given block.

Definition 90. 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

The response is the same as for the *Remote Read Request* message, respectively Definition 89.

7.5. Storage for Light Clients

The light client requires a persistent storage for saving the state of the blockchain. In addition, it requires efficient Serialization/De-serialization methods to transform SCALE (Section A.2.2.) encoded network traffic for storing and reading from the persistent storage.

8. 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 8.1.), whereas the relay chain validators only verify their validity (and later, their availability). It is possible that the collators of a parachain produce multiple parachain block candidates for a child of a specific block. Subsequently, they send the block candidates to the relay chain validators who are assigned to the specific parachain. The assignment is determined by the Runtime (Section 8.2.). Those validators are then required to check the validity of submitted candidates (Section 8.3.), then issue and collect statements (Section 8.2.1.) about the validity of candidates to other validators. This process is known as candidate backing. Once a candidate meets specified criteria for inclusion, the selected relay chain block author then chooses any of the backed candidates for each parachain and includes those into the relay chain block (Section 8.2.2.).

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 8.4.1.), then also collect the votes sent by other validators and include them in the relay chain state (Section 8.2.2.). 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 8.5.). If a validator does not have the candidate data, it must recover the candidate data (Section 8.4.2.).

8.1. Collations

Collations are proposed candidates Definition 121 to the Polkadot relay chain validators. The Polkodat network protocol is agnostic on what candidate production 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 8.3.). Collators, which are parachain nodes which produce candidate proposals and send them to the relay chain validator, must prepare pieces of data (Definition 91) in order to correctly comply with the requirements of the parachain protocol.

Definition 91. Collation

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

$$C = (M, H, R, h, P, p, w)$$

$$M = (u_n, \dots u_m)$$

$$H = (z_n, \dots z_m)$$

where

- M is an array of upward messages (Definition 127), u, interpreted by the relay chain itself.
- H is an array of outbound horizontal messages (Definition 129), z, interpreted by other parachains.
- R is an *Option* type (Definition 180) which can contain a parachain Runtime update. The new Runtime code is an array of bytes.
- h is the head data (Definition 123) produced as a result of execution of the parachain specific logic.
- P is the PoV block (Definition 122).

- p is an unsigned 32-bit integer indicating the number of processed downward messages (Definition 128).
- w is an unsigned 32-bit integer indicating the mark up to which all inbound HRMP messages have been processed by the parachain.

8.2. Candidate Backing

The Polkadot validator receives an arbitrary number of parachain candidates with associated proofs from untrusted collators. The assigned validators of each parachain (Definition 126) must verify and select a specific quantity of the proposed candidates and issue those as backable candidates to their 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 8.2.1.. Validators can retrieve information about assignments via the Runtime APIs Section C.9.2. respectively Section C.9.3..

8.2.1. Statements

The assigned validator checks the validity of the proposed parachains blocks (Section 8.3.) and issues *Valid* statements (Definition 92) 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 8.2.2.).

The validator issues validity statements votes in form of a validator protocol message (Definition 104).

Definition 92. Statement

A statement, S, is a data structure of the following format:

$$S = (d, A_i, A_s)$$

$$d = \begin{cases} 1 & \rightarrow & C_r \\ 2 & \rightarrow & C_h \end{cases}$$

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 95), 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.

8.2.2. Inclusion

The Polkadot validator includes the backed candidates as parachain inherent data (Definition 93) into a block as described Section 2.3.3.. 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 8.2.1.. The candidate approval process (Section 8.5.) ensures that only relay chain blocks are finalized where each candidate for each availability core meets the requirement of 2/3+ availability votes.

Definition 93. Parachain Inherent Data

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

$$I = (A, T, D, P_h)$$

$$T = (C_0, \dots C_n)$$

$$D = (d_n, \dots d_m)$$

$$C = (R, V, i)$$

$$V = (a_n, \dots a_m)$$

$$a = \begin{cases} 1 & \rightarrow & s \\ 2 & \rightarrow & s \end{cases}$$

$$A = (L_n, \dots L_m)$$

$$L = (b, v_i, s)$$

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 33).
- T is an array of backed candidates for including in the current block.
- D is an array of disputes.
- P_h is the parachain parent head data (Definition 123).
- d is a dispute statement (Section 8.7.2.1.).
- R is a committed candidate receipt (Definition 95).
- V is an array of validity votes themselves, expressed as signatures.
- i is a bitfield of indices of the validators within the validator group (Definition 126).
- 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 8.4.1.).
- v_i is the validator index of the authority set (Definition 33).

Definition 94. Candidate Receipt

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

$$R = (D, C_h)$$

where D is the candidate descriptor (Definition 96) and C_h is the hash of candidate commitments (Definition 97).

Definition 95. Committed Candidate Receipt

The committed candidate receipt, R, contains information about the candidate and the result of its execution that is included in the relay chain. This type is similar to the candidate receipt (Definition 94), but actually contains the execution results rather than just a hash of it. It's a data structure of the following format:

$$R = (D, C)$$

where D is the candidate descriptor (Definition 96) and C is the candidate commitments (Definition 97).

Definition 96. Candidate Descriptor

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

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

where

- p is the parachain Id (Definition 124).
- 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 220).
- 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 (Definition 123) of this candidate.
- R_h is the hash of the parachain Runtime.

Definition 97. 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 99). 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 (Definition 129) sent by the parachain.
- R is an *Option* value (Definition 180) that can contain a new parachain Runtime in case of an update.
- h is the parachain head data (Definition 123).
- p is an 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 an unsigned 32-bit integer indicating the watermark, which specifies the relay chain block number up to which all inbound horizontal messages have been processed.

8.3. Candidate Validation

Received candidates submitted by collators and must have their 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 220).
2. The signature of the collator is valid.
3. Validate the candidate by executing the parachain Runtime (Section 8.3.1.).

If all steps are valid, the Polkadot validator must create the necessary candidate commitments (Definition 97) and submit the appropriate statement for each candidate (Section 8.2.1.).

8.3.1. 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 C.9.8. and Section C.9.9. respectively. The retrieved parachain Runtime might need to be decompressed based on the magic identifier as described in Section 8.3.2..

In order to validate a parachain block, the Polkadot validator must prepare the validation parameters (Definition 98), then use its local Wasm execution environment (Section 2.6.3.) 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 99).

Definition 98. Validation Parameters

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

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

where

- h is the parachain head data (Definition 123).
- b is the block body (Definition 122).
- B_i is the latest relay chain block number.
- S_r is the relay chain block storage root (Section 2.4.4.).

Definition 99. 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 97), which then will be inserted into the relay chain via the parachain inherent data (Definition 93). The validation result, V, is a data structure of the following format:

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

$$M_u = (m_0, \dots m_n)$$

$$M_h = (t_0, \dots t_n)$$

where

- h is the parachain head data (Definition 123).
- R is an *Option* value (Definition 180) 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 (Definition 129) sent by the parachain.
- p is an 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 an unsigned 32-bit integer indicating the watermark, which specifies the relay chain block number up to which all inbound horizontal messages have been processed.

8.3.2. Runtime Compression

	Runtime compression is not documented yet.

8.4. Availability

8.4.1. Availability Votes

The Polkadot validator must issue a bitfield (Definition 131) 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 the form of parachain inherent data (Section 8.2.2.) by a block author. A validator can retrieve that data by calling the appropriate Runtime API entry (Section C.9.3.), then create a bitfield indicating for which candidate the validator has availability data stored and broadcast it to the network (Definition 108). When sending the bitfield distribution message, the validator must ensure B_h is set appropriately, 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 8.4.2.. 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 8.5.) must be kept by validators for about 25 hours since disputes (Section 8.6.) can occur and the candidate needs to be checked again.

The validator issues availability votes in form of a validator protocol message (Definition 105).

8.4.2. 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 8.2.) respectively whether the candidate should be approved (Section 8.5.). Additionally, peers can send availability requests as defined in Definition 112 and Definition 114 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 A.4.1.). 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 C.9.13..

8.5. 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 8.2.2.), which can be retrieved via the Runtime API (Section C.9.3.), 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 to prevent the finalization of invalid blocks. If an honest validator detects an invalid block that was approved by one or more validators, the honest validator must issue a dispute which will cause escalations, resulting in consequences for all malicious parties, i.e., slashing. This mechanism is described more in Section 8.5.1..

8.5.1. 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 101), 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 8.5.2.).

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 109), 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 8.4.2.) and then validate the candidate (Section 8.3.).

The validator issues approval votes in form of a validator protocol message (Definition 104) respectively disputes (Section 8.6.).

8.5.2. 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 candidate have been proposed, which can be retrieved via the Runtime API (Section C.9.3.), 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 8.5.1., 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 C.9.13. 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 C.9.13., 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 100. 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 in the following way:

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

where

- Transcript constructs a VRF transcript (Definition 165).
- b_r is the BABE randomness of the current epoch (Definition 67).
- b_s is the current BABE slot (Definition 50).
- e_i is the current BABE epoch index (Definition 50).
- A is the public key of the authority.

Definition 101. Availability Core VRF Assignment

An availability core VRF assignment is computed by a relay chain validator to determine which availability core (Definition 125) a validator is assigned to and should vote for approvals. Computing this assignment relies on the VRF mechanism, transcripts, and STROBE operations described further in Section A.1.3..

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 C.9.13.). 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:

$$t_1 \leftarrow \text{Transcript}(\text{'A\&V MOD'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{'RC-VRF'}, R_s)$$

$$t_3 \leftarrow \text{append}(t_2, \text{'sample'}, s)$$

$$t_4 \leftarrow \text{append}(t_3, \text{'vrf-nm-pk'}, p_k)$$

$$t_5 \leftarrow \text{meta-ad}(t_4, \text{'VRFHash'}, \text{False})$$

$$t_6 \leftarrow \text{meta-ad}(t_5, 64_{\text{le}}, \text{True})$$

$$i \leftarrow \text{prf}(t_6, \text{False})$$

$$o = s_k \cdot i$$

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 100. Following:

$$t_1 \leftarrow \text{Transcript}(\text{'VRFResult'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{''}, \text{'A\&V CORE'})$$

$$t_3 \leftarrow \text{append}(t_2, \text{'vrf-in'}, i)$$

$$t_4 \leftarrow \text{append}(t_3, \text{'vrf-out'}, o)$$

$$t_5 \leftarrow \text{meta-ad}(t_4, \text{''}, \text{False})$$

$$t_6 \leftarrow \text{meta-ad}(t_5, 4_{\text{le}}, \text{True})$$

$$r \leftarrow \text{prf}(t_6, \text{False})$$

$$c_i = r \bmod a_c$$

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 C.9.13.). The resulting integer, c_i, indicates the parachain Id (Definition 124). If the parachain Id doesn't exist, as can be retrieved by the Runtime API (Section C.9.3.), the validator discards that value and continues with the next iteration. If the Id does exist, the validator continues with the following steps:

$$t_1 \leftarrow \text{Transcript}(\text{'A\&V ASSIGNED'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{'core'}, c_i)$$

$$p \leftarrow \text{dleq\_prove}(t_2, i)$$

where dleq_prove is described in Definition 162. The resulting values of o, p and s are used to construct an assignment certificate (Definition 103) of kind *0*.

Definition 102. 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 8.5.2.. Computing this assignment relies on the VRF mechanism, transcripts, and STROBE operations described further in Section A.1.3..

The validator executes the following steps:

$$t_1 \leftarrow \text{Transcript}(\text{'A\&V DELAY'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{'RC-VRF'}, R_s)$$

$$t_3 \leftarrow \text{append}(t_2, \text{'core'}, c_i)$$

$$t_4 \leftarrow \text{append}(t_3, \text{'vrf-nm-pk'}, p_k)$$

$$t_5 \leftarrow \text{meta-ad}(t_4, \text{'VRFHash'}, \text{False})$$

$$t_6 \leftarrow \text{meta-ad}(t_5, 64_{\text{le}}, \text{True})$$

$$i \leftarrow \text{prf}(t_6, \text{False})$$

$$o = s_k \cdot i$$

$$p \leftarrow \text{dleq\_prove}(t_6, i)$$

The resulting value p is the VRF proof (Definition 161). dleq_prove is described in Definition 162.

The tranche, d, is determined as:

$$t_1 \leftarrow \text{Transcript}(\text{'VRFResult'})$$

$$t_2 \leftarrow \text{append}(t_1, \text{''}, \text{'A\&V TRANCHE'})$$

$$t_3 \leftarrow \text{append}(t_2, \text{'vrf-in'}, i)$$

$$t_4 \leftarrow \text{append}(t_3, \text{'vrf-out'}, o)$$

$$t_5 \leftarrow \text{meta-ad}(t_4, \text{''}, \text{False})$$

$$t_6 \leftarrow \text{meta-ad}(t_5, 4_{\text{le}}, \text{True})$$

$$c \leftarrow \text{prf}(t_6, \text{False})$$

$$d = d \mathrm{mod} (d_c + d_z) - d_z$$

where

- d_c is the number of delayed tranches by total as specified by the session info, retrieved via the Runtime API (Section C.9.13.).
- d_z is the zeroth delay tranche width as specified by the session info, retrieved via the Runtime API (Section C.9.13.).

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 103) of kind *1*.

Definition 103. 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 101. This certificate contains the computed VRF output and is a data structure of the following format:

$$(k, o, p)$$

$$k = \begin{cases} 0 & \rightarrow & s \\ 1 & \rightarrow & c_i \end{cases}$$

where k indicates the kind of the certificate, respectively the value *0* proves the availability core assignment (Definition 101), followed by the sample number s, and the value *1* proves the delayed availability core assignment (Definition 102), followed by the core index c_i (Section C.9.3.). o is the VRF output and p is the VRF proof.

8.6. Disputes

> **INFO**
> Disputes are not documented yet.

8.7. Network Messages

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

8.7.1. 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 A.2.2.).

Definition 104. 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 8.2.) and the approval process (Section 8.5.). The validator protocol message, M, is a varying datatype of the following format:

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

where

- M_f is a bitfield distribution message (Definition 108).
- M_s is a statement distribution message (Definition 107).
- M_a is a approval distribution message (Definition 109).

Definition 105. Collation Protocol Message

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

$$M = \{0 \quad \rightarrow \quad M_c$$

where M_c is the collator message (Definition 106).

Definition 106. Collator Message

The collator message is sent as part of the collator protocol message (Definition 105). The collator message, M, is a varying datatype of the following format:

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

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 (Definition 124).
- C_s is the signature of the collator using the *PeerId* of the collators node.
- H is the hash of the parachain block (Definition 122).
- S is a full statement (Definition 92).

Definition 107. Statement Distribution Message

The statement distribution message is sent as part of the validator protocol message (Definition 105) indicates the validity vote of a validator for a given candidate, described further in Section 8.2.1.. The statement distribution message, M, is of varying type of the following format:

$$M = \begin{cases} 0 & \rightarrow & (B_h, S) \\ 1 & \rightarrow & S_m \end{cases}$$

$$S_m = (B_h, C_h, A_i, A_s)$$

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 118).
- B_h is the hash of the relay chain parent, indicating the state this message is for.
- S is a full statement (Definition 92).
- A_i is the validator index in the authority set (Definition 33) that signed this message.
- A_s is the signature of the validator.

Definition 108. Bitfield Distribution Message

The bitfield distribution message is sent as part of the validator protocol message (Definition 104) and indicates the availability vote of a validator for a given candidate, described further in Section 8.4.1.. This message is sent in the form of a validator protocol message (Definition 104). The bitfield distribution message, M, is a datastructure of the following format:

$$M = \{0 \quad \rightarrow \quad (B_h, P)$$

$$P = (d, A_i, A_s)$$

where

- B_h is the hash of the relay chain parent, indicating the state this message is for.
- d is the bitfield array (Definition 131).
- A_i is the validator index in the authority set (Definition 33) that signed this message.
- A_s is the signature of the validator.

Definition 109. Approval Distribution Message

The approval distribution message is sent as part of the validator protocol message (Definition 104) and indicates the approval vote of a validator for a given candidate, described further in Section 8.5.1.. The approval distribution message, M, is a varying datatype of the following format:

$$M = \begin{cases} 0 \quad \rightarrow \quad ((C, I)_0 \dots (C, I)_n) \\ 1 \quad \rightarrow \quad (V_0, \dots V_n) \end{cases}$$

$$C = (B_h, A_i, c_a)$$

$$c_a = (c_k, P_o, P_p)$$

$$c_k = \begin{cases} 0 \rightarrow s \\ 1 \rightarrow i \end{cases}$$

$$V = (B_h, I, A_i, A_s)$$

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 that 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 to the order of the availability cores (Section C.9.3.).
- B_h is the relay chain block hash where the candidate appears.
- A_i is the authority set Id (Definition 69) 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 8.5..
- P_o is a VRF output and P_p its corresponding proof.

8.7.2. 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 110. 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 data structure 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 111.

Definition 111. PoV Fetching Response

The PoV fetching response is sent by nodes to the clients who issued a PoV fetching request (Definition 110). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & \rightarrow & B \\ 1 & \rightarrow & \phi \end{cases}$$

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

Definition 112. Chunk Fetching Request

The chunk fetching request is sent by clients who want to retrieve chunks of a parachain candidate. The request is a data structure 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 113.

Definition 113. Chunk Fetching Response

The chunk fetching response is sent by nodes to the clients who issued a chunk fetching request (Definition 112). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & \rightarrow & C_r \\ 1 & \rightarrow & \phi \end{cases}$$

$$C_r = (c, c_p)$$

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 $(b_n \ldots b_m)$.

Definition 114. 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 data structure 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 115.

Definition 115. Available Data Response

The available data response is sent by nodes to the clients who issued an available data request (Definition 114). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & \rightarrow & A \\ 1 & \rightarrow & \phi \end{cases}$$

$$A = (P_{ov}, D_{pv})$$

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 (Definition 122) and D_{pv} is the persisted validation data (Definition 220).

Definition 116. 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 data structure 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 (Definition 124). The response message is defined in Definition 117.

Definition 117. Collation Fetching Response

The collation fetching response is sent by nodes to the clients who issued a collation fetching request (Definition 116). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & \rightarrow & (C_r, B) \end{cases}$$

where 0 is followed by the candidate receipt (Definition 94), C_r, as and the PoV block (Definition 122), B. This type does not notify the client about a statement that was not found.

Definition 118. Statement Fetching Request

The statement fetching request is sent by clients who want to retrieve statements about a given candidate. The request is a data structure 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 95). The response message is defined in Definition 119.

Definition 119. Statement Fetching Response

The statement fetching response is sent by nodes to the clients who issued a collation fetching request (Definition 118). The response, R, is a varying datatype of the following format:

$$R = \begin{cases} 0 & \rightarrow & C_r \end{cases}$$

where C_r is the committed candidate receipt (Definition 95). No response is returned if no statement is found.

8.7.2.1. Dispute Request

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

$$D_r = (C_r, S_i, I_v, V_v)$$

$$I_v = (A_i, A_s, k_i)$$

$$V_v = (A_i, A_s, k_v)$$

$$k_i = \begin{cases} 0 & \rightarrow & \phi \end{cases}$$

$$k_v = \begin{cases} 0 & \rightarrow & \phi \\ 1 & \rightarrow & C_h \\ 2 & \rightarrow & C_h \\ 3 & \rightarrow & \phi \end{cases}$$

where

- C_r is the candidate that is being disputed. The structure is a candidate receipt (Definition 94).
- 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 33).
- 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 8.7.2.2..

8.7.2.2. Dispute Response

The dispute response is sent by nodes to the clients who issued a dispute request (Section 8.7.2.1.). The response, R, is a varying type of the following format:

$$R = \{0 \quad \rightarrow \quad \phi$$

where 0 indicates that the dispute was successfully processed.

8.8. Definitions

Definition 120. Collator

> A collator is a parachain node that sends parachain blocks, known as candidates (Definition 121), to the relay chain validators. The relay chain validators are not concerned with how the collator works or how it creates candidates.

Definition 121. Candidate

> A candidate is a submitted parachain block (Definition 122) to the relay chain validators. A parachain block stops being referred to as a candidate as soon it has been finalized.

Definition 122. Parachain Block

> A parachain block or a Proof-of-Validity block (PoV block) contains the necessary data for the 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.

Definition 123. Head Data

The head data contains information about a parachain block (Definition 122). 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 array.

Definition 124. Parachain Id

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

Definition 125. Availability Core

Availability cores are slots used to process parachains. The Runtime assigns each parachain to an availability core, and validators can fetch information about the cores, such as parachain block candidates, by calling the appropriate Runtime API (Section C.9.3.). Validators are not concerned with the internal workings from the Runtimes perspective.

Definition 126. Validator Groups

Validator groups indicate which validators are responsible for creating backable candidates for parachains (Section 8.2.), and are assigned by the Runtime (Section C.9.2.). Validators are not concerned with the internal workings from the Runtimes perspective. Collators can use this information for submitting blocks.

Definition 127. Upward Message

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

Definition 128. Downward Message

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

Definition 129. Outbound HRMP Message

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

$$(I, M)$$

where I is the recipient Id (Definition 124) and M is an upward message (Definition 127).

Definition 130. Inbound HRMP Message

An inbound HRMP message (Horizontal Relay-routed Message Passing) is seen from the perspective of a recipient parachain sent from another parachain by passing it through the relay chain. It's a data structure 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 (Definition 128).

Definition 131. Bitfield Array

A bitfield array contains single-bit values, which indicates whether a candidate is available. The number of items is equal to the number of availability cores (Definition 125), 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 95) and that the validator has a stored chunk of the parachain block (Definition 122).

Polkadot Runtime

Description of various useful Runtime internals

📄 9. Extrinsics

9.1. Introduction

📄 10. Weights

10.1. Motivation

📄 11. Consensus

11.1. BABE digest messages

📄 12. Metadata

The runtime metadata structure contains all the information necessary on how to interact with the Polkadot runtime. Considering that Polkadot runtimes are upgradable and, the…

9. Extrinsics

9.1. 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.

9.2. Preliminaries

Definition 132. Extrinsic

An extrinsic , tx, is a tuple consisting of the extrinsic version, T_v (Definition 133), 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 9.3.1..

Definition 133. 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 versions are rejected.

The most significant 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, a signed extrinsic represents T_v as `132` while an unsigned extrinsic represents it as `4`.

9.3. Extrinsics Body

9.3.1. 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

- A_i: the 32-byte address of the sender (Definition 134).
- Sig: the signature of the sender (Definition 135).
- E: the extra data for the extrinsic (Definition 136).
- M_i: the indicator of the Polkadot module (Definition 137).
- $F_i(m)$: the indicator of the function of the Polkadot module (Definition 138).

Definition 134. 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 135. 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:

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

Signature types vary in size, 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.

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

where

- M_i: the module indicator (Definition 137).
- $F_i(m)$: the function indicator of the module (Definition 138).
- E: the extra data (Definition 136).
- R_v: a UINT32 containing the specification version (`spec_version`) of the Runtime (Section C.4.1.), which can be updated and is therefore subject to change.
- F_v: a UINT32 containing the transaction version (`transaction_version`) of the Runtime (Section C.4.1.), which can be updated and is therefore subject to change.
- $H_h(G)$: 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 139.

Definition 136. Extra Data

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

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

where

- T_{mor}: contains the SCALE encoded mortality of the extrinsic (Definition 139).
- 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 137. 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 138. 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:

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

9.3.2. Mortality

Definition 139. 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 9.3.2.2..

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 135). 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 136, in the SCALE encoded form of T_{mor} (Section 9.3.2.2.). 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".

9.3.2.1. 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`.

9.3.2.2. 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 a 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 Encode Mortality and Decode Mortality.

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

Algorithm 25. Encode Mortality

Algorithm Encode Mortality

Require: M_{per}, M_{pha}

1: **return** 0 **if** *extrinsic is immortal*
2: **init** $factor = \text{LIMIT}(M_{per} \gg 12, 1, \phi)$
3: **init** $left = \text{LIMIT}(\text{TZ}(M_{per}) - 1, 1, 15)$
4: **init** $right = \frac{M_{pha}}{factor} \ll 4$
5: **return** $left | right$

Algorithm 26. Decode Mortality

Algorithm Decode Mortality

Require: T_{mor}

1: **return** $Immortal$ **if** $T^{b0}_{mor} = 0$
2: **init** $enc = T^{b0}_{mor} + (T^{b1}_{mor} \ll 8)$
3: **init** $M_{per} = 2 \ll (enc \bmod (1 \ll 4))$
4: **init** $factor = \text{LIMIT}(M_{per} \gg 12, 1, \phi)$
5: **init** $M_{pha} = (enc \gg 4) * factor$
6: **return** (M_{per}, M_{pha})

where

- $T^{b0}_{\{mor\}}$: 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.
- $\gg$: performs a binary right shift operation.
- $\ll$: performs a binary left shift operation.
- $|$: performs a bitwise OR operation.

10. Weights

10.1. Motivation

The Polkadot network, like any other permissionless system, needs to implement a mechanism to measure and limit the usage in order to establish an economic incentive structure, prevent network overload, and 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 10.2.1.). Moreover, the weight system can be used to calculate a fee for executing each extrinsics according to its weight (as described in Section 10.6.1.).

Additionally, Polkadot introduces a specified block ratio (as defined in Section 10.2.1.), 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 of Polkadot itself.

To begin, we introduce in Section 10.2. the assumption upon which the Polkadot transaction weight system is designed. In Section 10.2.1., we discuss the limitation Polkadot needs to enforce on the block size. In Section 10.3., we describe in detail the procedure upon which the weight of any transaction should be calculated. In Section 10.5., we present how we apply this procedure to compute the weight of particular runtime functions.

10.2. 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 that 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 140 and Definition 143:

Definition 140. 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 141. 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 142. 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 143. 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 144. Weight Function

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

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

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 2.3.3. 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:

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

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.

10.2.1. Limitations

In this section, we discuss how applying the limitation defined in Definition 143 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 its block length in Definition 145:

Definition 145. 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:

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

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 an 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.

10.3. 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 of 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 10.4.1..

The general algorithm to calculate $\mathcal{W}(E)$ is described in the Section 10.4..

10.4. Benchmarking

Calculating the extrinsic weight solely based on the 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 certainly 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 10.4.1.. 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 10.5.) covers the analysis process and the implementation of preliminary work in more detail.

10.4.1. 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.

10.4.1.1. 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 depends 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 146.
- **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 146.

Definition 146. 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 10.2.1..

10.4.2. Parameters

The input 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 that might decelerate or heavily impact the 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 10.4.1.1.. 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 between theoretical and practical usage, the author reserves the right to make adjustments to the input parameters and assign weights according to the observed behavior of the actual, real-world network.

10.4.2.1. 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%.

10.4.3. 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.

10.4.4. 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.

10.5. Practical examples

This section walks through Runtime functions available in the Polkadot Runtime to demonstrate the analysis process as described in Section 10.4.1..

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.

10.5.1. 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 on-chain, directly in the corresponding Identity Info. It's also noteworthy 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 judgment 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 judgment fee varies for each registrar.

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

10.5.1.1. 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));
```

10.5.1.2. 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, therefore increasing its footprint in the state storage. Additionally, if a new value gets inserted into the byte array, memory gets 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.

10.5.1.3. 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 27. `request_judgement` **Runtime Function Benchmark**

Algorithm "request_judgement"` Runtime function benchmark

Ensure: $\mathcal{W}$
1: **init** $collection = \{\}$
2: **for** $amount \leftarrow 1, MaxRegistrars$ **do**
3: GENERATE-REGISTRARS($amount$)
4: $caller \leftarrow$ CREATE-ACCOUNT($caller, 1$)
5: SET-BALANCE($caller, 100$)
6: $time \leftarrow$ TIMER(REQUEST-JUDGEMENT(RANDOM($amount$), 100))
7: ADD-TO($collection, time$)
8: **end for**
9: $\mathcal{W} \leftarrow$ COMPUTE-WEIGHT($collection$)
10: **return** $\mathcal{W}$

where

- Generate-Registrars($amount$)

  Creates a 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 an account. This function only creates an address and does not conduct any I/O.

- Set-Balance($amount$, $balance$)

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

- Timer($function$)

  Measures the time from the start of the specified function 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).

10.5.2. Practical Example #2: `payout_stakers`

10.5.2.1. 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 operations for each of them.

First, this function makes a 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 information 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 receive 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 loop 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()));
  }
}
```

10.5.2.2. 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.

10.5.2.3. Benchmarking Framework

Definition 147. History Depth

History 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 148. Maximum Nominator Reward

Maximum 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 28. `payout_stakers` **Runtime Function Benchmark**

Algorithm "payout_stakers"` Runtime function benchmark

Ensure: $\mathcal{W}$

1: **init** $collection = \{\}$
2: **for** $amount \leftarrow 1, MaxNominatorRewardedPerValidator$ **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
```

where

- Generate-Validator()

  Creates a validator 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 function 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).

10.5.3. Practical Example #3: `transfer`

The $transfer$ function of the `balances` module is designed to move the specified balance by the sender to the receiver.

10.5.3.1. 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 needs to pay close attention to the property `AllowDeath` and to how the function treats existings and non-existing accounts differently. Two types of behaviors are to consider:

- If the transfer completely depletes the sender account balance to zero (or below the minimum "keep-alive" requirement), it removes the address and all associated data from storage.
- If the recipient account has no balance, the transfer also needs to create the recipient account.

10.5.3.2. 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 "transfer" Runtime function benchmark 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 themselves 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 "transfer" Runtime function benchmark.

10.5.3.3. Benchmarking Framework

The benchmarking implementation for the Polkadot Runtime function $transfer$ is defined as follows (starting with the Main function):

Algorithm 29. `transfer` Runtime Function Benchmark

Algorithm "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
```

13: $\mathcal{W} \leftarrow$ Compute-Weight($collection$)
14: **return** $\mathcal{W}$
15: **end function**
16: **function** Run-Benchmark($index$, $balance$)
17: $sender \leftarrow$ Create-Account($caller$, $index$)
18: $recipient \leftarrow$ Create-Accouny($recipient$, $index$)
19: Set-Balance($sender$, $balance$)
20: $time \leftarrow$ Timer(Transfer($sender$, $recipient$, $balance$))
21: **return** $time$
22: **end function**

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).

10.5.4. 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.

10.5.4.1. Analysis

Similarly to the `payout_stakers` function (Section 10.5.2.), 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));
```

10.5.4.2. 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.

10.5.4.3. 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.

10.5.4.4. Benchmarking Framework

The benchmarking implementation for the Polkadot Runtime function `withdraw_unbonded` is defined as follows:

Algorithm 30. `withdraw_unbonded` **Runtime Function Benchmark**

Algorithm "withdraw_unbonded" Runtime function benchmark

Ensure: $\mathcal{W}$

function MAIN()
  init $collection = \{\}$
  for $balance \leftarrow 1, 100$ **do**
    $stash \leftarrow$ CREATE-ACCOUNT$(stash, 1)$
    $controller \leftarrow$ CREATE-ACCOUNT$(controller, 1)$
    SET-BALANCE$(stash, 100)$
    SET-BALANCE$(controller, 1)$
    BOND$(stash, controller, balance)$
    PASS-ERA()
    UNBOND$(controller, balance)$
    PASS-ERA()
    $time \leftarrow$ TIMER(WITHDRAW-UNBONDED$(controller)$)
    ADD-TO$(collection, time)$
  end for
  $\mathcal{W} \leftarrow$ COMPUTE-WEIGHT$(collection)$
  return $\mathcal{W}$
end function

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($amount$, $balance$)

  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).

10.6. 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.

10.6.1. 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:

$$\begin{aligned} fee = & \ base\ fee \\ & + \text{length of transaction in bytes} \times \text{length fee} \\ & + \text{weight to fee} \end{aligned}$$

10.6.2. 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.

10.6.3. 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:

$$finalfee = fee \times FeeMultiplier$$

10.6.3.1. 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:

$$\begin{aligned} diff = & \quad (target\ weight - previous\ block\ weight) \\ v = & \quad 0.00004 \\ next\ weight = & \quad weight \times (1 + (v \times diff) + (v \times diff)^2/2) \end{aligned}$$

Polkadot defines the `target_weight` as 0.25 (25%). More information about this algorithm is described in the Web3 Foundation research paper.

11. Consensus

11.1. 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 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 randomness seed is described in Epoch-Randomness, 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 31. Epoch Randomness

Algorithm Epoch-Randomness

Require: $n > 2$

init $\rho \leftarrow \phi$
for B in SubChain($\mathcal{E}_{n-2}$) **do**
    $\rho \leftarrow \rho || d_B$
end for
return Blake2b(Epoch-Randomness$(n-1)||n||\rho$)

where n is the epoch index.

12. Metadata

The runtime metadata structure contains all the information necessary on how to interact with the Polkadot runtime. Considering that Polkadot runtimes are upgradable and, therefore, any interfaces are subject to change, the metadata allows developers to structure any extrinsics or storage entries accordingly.

The metadata of a runtime is provided by a call to `Metadata_metadata` (Section C.5.1.) and is returned as a scale encoded (Section A.2.2.) binary blob. How to interpret and decode this data is described in this chapter.

12.1. Structure

The Runtime Metadata is a data structure of the following format:

$$(M, v_m, R, P, t_e, v_e, E, t_r)$$

$$R = (r_0, \ldots, r_n)$$

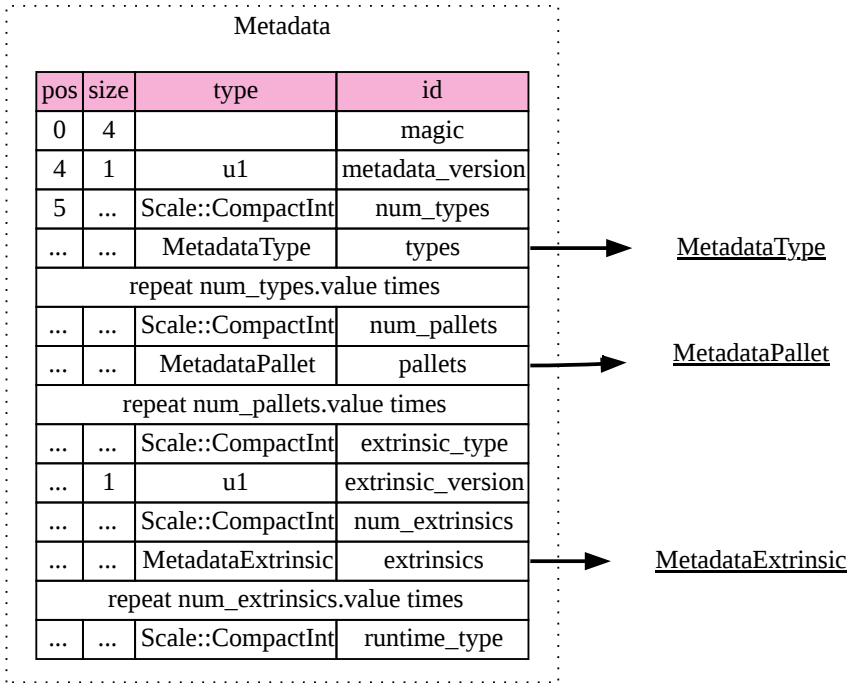
$$P = (p_0, \ldots, p_n)$$

$$E = (e_0, \ldots, e_n)$$

where

- M are the first four constant bytes, spelling "meta" in ASCII.
- v_m is an unsigned 8-bit integer indicating the format version of the metadata structure (currently the value of `14`).
- R is a sequence (Definition 182) of type definitions r_i (Definition 149).
- P is a sequence (Definition 182) of pallet metadata p_i (Section 12.2.).
- t_e is the type Id (Definition 150) of the extrinsics.
- v_e is an unsigned 8-bit integer indicating the format version of the extrinsics (implying a possible breaking change).
- E is a sequence (Definition 182) of extrinsics metadata e_i (Definition 160).
- t_r is the type Id (Definition 150) of the runtime.

Image 8. Metadata



Definition 149. Runtime Registry Type Entry

A registry entry contains information about a type in its portable form for serialization. The entry is a data structure of the following format:

$$r_i = (\mathrm{id}_t, p, T, D, c)$$

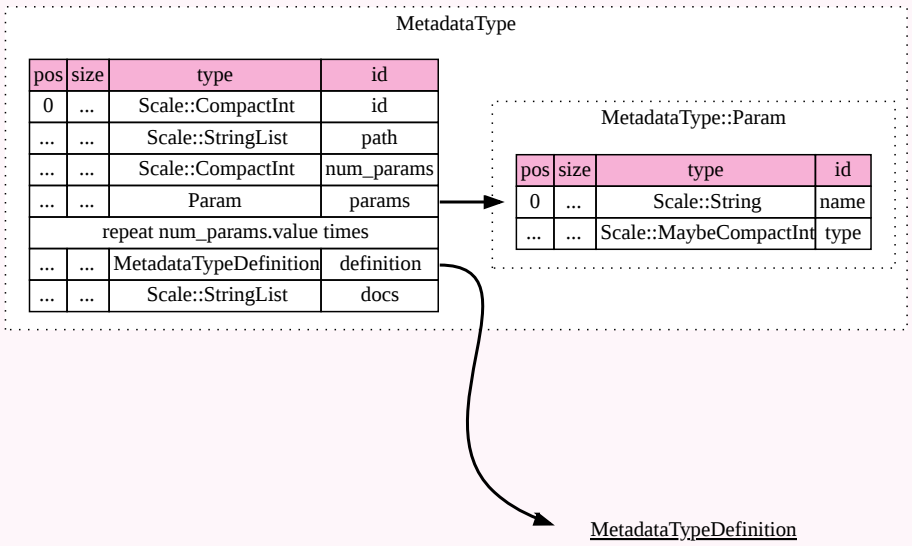
$$T = (t_0, \ldots, t_n)$$

$$t_i = (n, y)$$

where

- id_t is a compact integer indicating the identifier of the type.
- p is the path of the type, optional and based on the source file location. Encoded as a sequence (Definition 182) of strings.
- T is a sequence (Definition 182) of generic parameters (empty for non-generic types).
  - n is the name string of the generic type parameter
  - y is a *Option* type containing a type Id (Definition 150).
- D is the type definition (Definition 151).
- c is the documentation as sequence (Definition 182) of strings.

Image 9. Metadata Type



Definition 150. Runtime Type Id

The **runtime type Id** is a compact integer representing the index of the entry (Definition 149) in R, P or E of the runtime metadata structure (Section 12.1.), depending on context (starting at 0).

Definition 151. Type Variant

The type definition D is a varying datatype (Definition 178) and indicates all the possible types of encodable values a type can have.

$$D = \begin{cases} 0 \rightarrow C & \text{composite type (e.g. structure or tuple)} \\ 1 \rightarrow V & \text{variant type} \\ 2 \rightarrow s_v & \text{sequence type varying length} \\ 3 \rightarrow S & \text{sequence with fixed length} \\ 4 \rightarrow T & \text{tuple type} \\ 5 \rightarrow P & \text{primitive type} \\ 6 \rightarrow e & \text{compact encoded type} \\ 7 \rightarrow B & \text{sequence of bits} \end{cases}$$

where

- C is a sequence of the following format:

  $C = (f_0, \ldots, f_n)$

  - f_i is a field (Definition 152).
- V is a sequence of the following format:

  $V = (v_0, \ldots, v_n)$

  - v_i is a variant (Definition 153).
- s_v is a type Id (Definition 150).
- S is of the following format:

  $S = (l, y)$

  - l is an unsigned 32-bit integer indicating the length
  - y is a type Id (Definition 150).
- T is a sequence (Definition 182) of type Ids (Definition 150).
- P is a varying datatype (Definition 178) of the following structure:

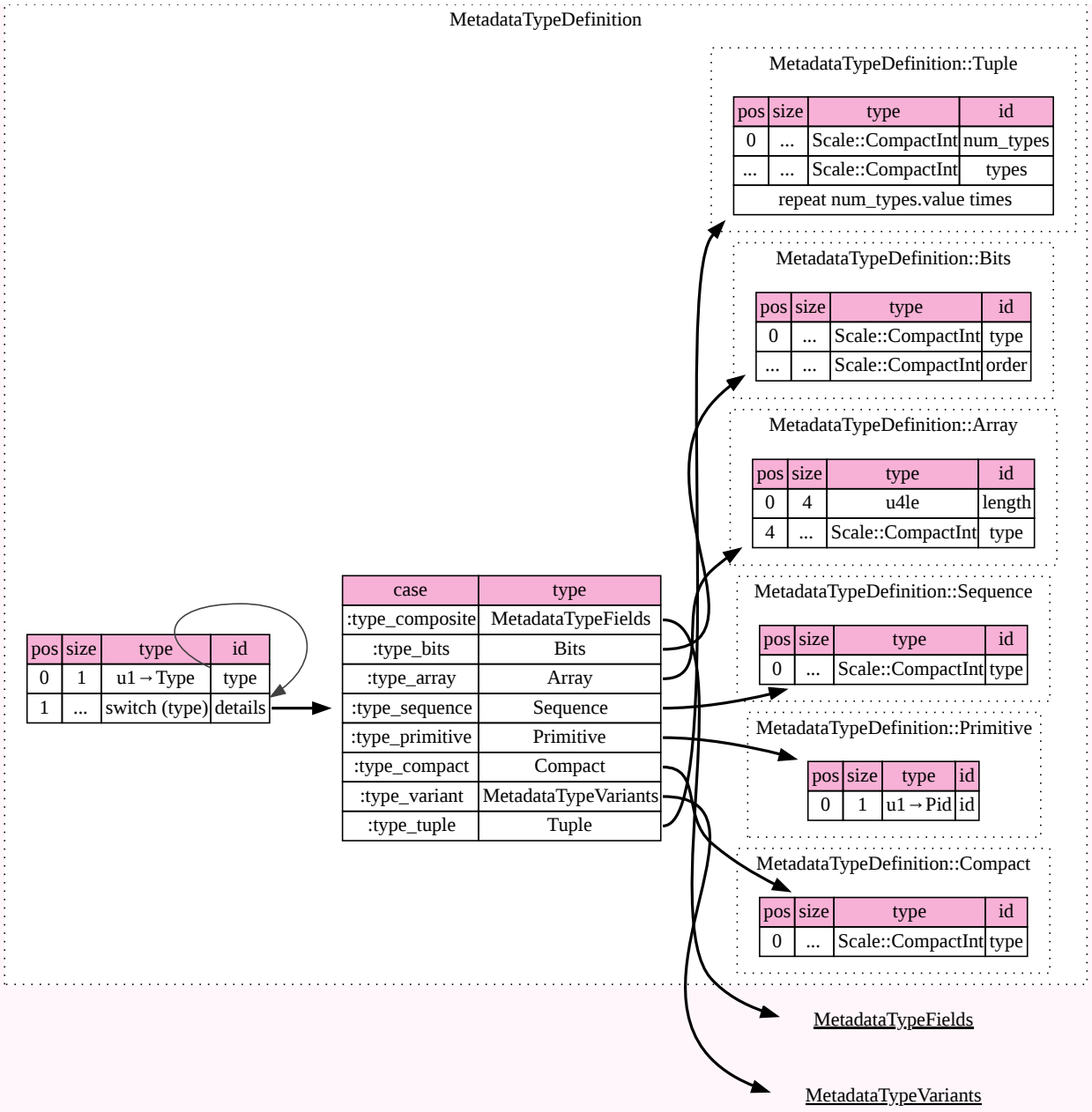
$$P = \begin{cases} 0 & \text{boolean} \\ 1 & \text{char} \\ 2 & \text{string} \\ 3 & \text{unsigned 8-bit integer} \\ 4 & \text{unsigned 16-bit integer} \\ 5 & \text{unsigned 32-bit integer} \\ 6 & \text{unsigned 64-bit integer} \\ 7 & \text{unsigned 128-bit integer} \\ 8 & \text{unsigned 256-bit integer} \\ 9 & \text{signed 8-bit integer} \\ 10 & \text{signed 16-bit integer} \\ 11 & \text{signed 32-bit integer} \\ 12 & \text{signed 64-bit integer} \\ 13 & \text{signed 128-bit integer} \\ 14 & \text{signed 256-bit integer} \end{cases}$$

- e is a type Id (Definition 150).
- B is a data structure of the following format:

  $B = (s, o)$

  - s is a type Id (Definition 150) representing the bit store order (external reference)
  - o is a type Id (Definition 150) the bit order type (external reference).

Image 10. Metadata Type Definition



Definition 152. Field

A field of a data structure of the following format:

$$f_i = (n, y, y_n, C)$$

where

- n is an *Option* type containing the string that indicates the field name.
- y is a type Id (Definition 150).
- y_n is an *Option* type containing a string that indicates the name of the type as it appears in the source code.
- C is a sequence of varying length containing strings of documentation.

Image 11. Metadata Type Fields

MetadataTypeFields

pos	size	type	id
0	...	Scale::CompactInt	num_fields
...	...	Field	fields
repeat num_fields.value times			

MetadataTypeFields::Field

pos	size	type	id
0	...	Scale::MaybeString	name
...	...	Scale::CompactInt	type
...	...	Scale::MaybeString	typename
...	...	Scale::StringList	docs

Definition 153. Variant

A struct variant of the following format:

$$v_i = (n, F, k, C)$$

where

- n is a string representing the name of the variant.
- F is a possible empty array of varying length containing field (Definition 152) elements.
- k is an unsigned 8-bit integer indicating the index of the variant.
- C is a sequence of strings containing the documentation.

Image 12. Metadata Type Variants

MetadataTypeVariants

pos	size	type	id
0	...	Scale::CompactInt	num_variants
...	...	Variant	variants
repeat num_variants.value times			

MetadataTypeVariants::Variant

pos	size	type	id
0	...	Scale::String	name
...	...	MetadataTypeFields	composite
...	1	u1	index
...	...	Scale::StringList	docs

MetadataTypeFields

12.2. Pallet Metadata

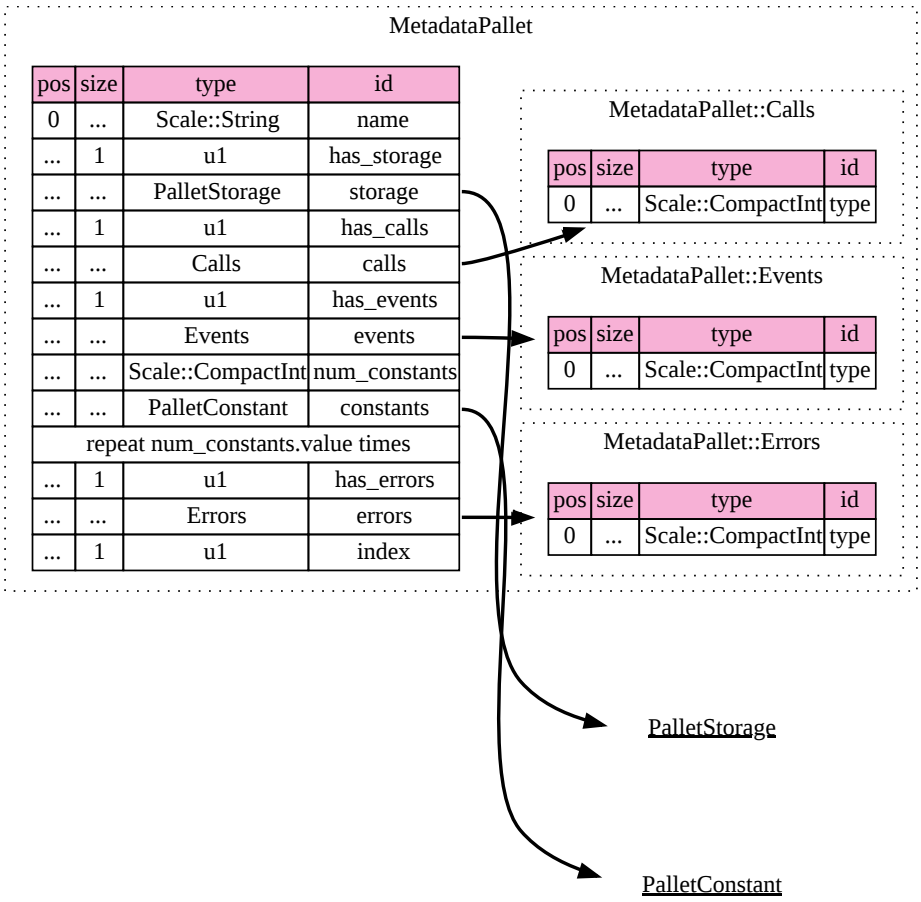
All the metadata about a pallet, part of the main structure (Section 12.1.) and of the following format:

$$p_i = (n, S, a, e, C, e, i)$$

where

- n is a string representing the pallet name.
- S is an *Option* type containing the pallet storage metadata (Definition 154).
- a is an *Option* type (Definition 180) containing the type Id (Definition 150) of pallet calls.
- e is an *Option* type (Definition 180) containing the type Id (Definition 150) of pallet events.
- C is an *Sequence* (Definition 182) of all pallet constant metadata (Definition 159).
- e is an *Option* type (Definition 180) containing the type Id (Definition 150) of the pallet error.
- i is an unsigned 8-bit integer indicating the index of the pallet, which is used for encoding pallet events and calls.

Image 13. Metadata Pallet



Definition 154. Pallet Storage Metadata

The metadata about pallets storage.

$$S = (p, E)$$

$$E = (e_0, \ldots, e_n)$$

where

- p is the string representing the common prefix used by all storage entries.
- E is an array of varying lengths containing elements of storage entries (Definition 155).

Definition 155. Storage Entry Metadata

The metadata about a pallets storage entry.

$$e_i = (n, m, y, d, C)$$

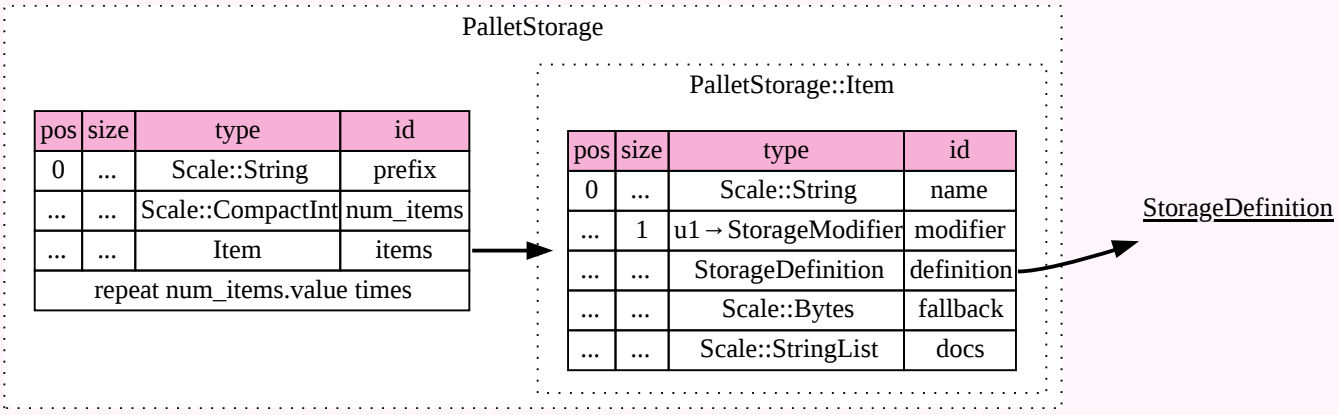
$$C = (c_0, \ldots, c_n)$$

where

- n is the string representing the variable name of the storage entry.
- m is an enum type determining the storage entry modifier (Definition 156).
- y is the type of the value stored in the entry (Definition 157).
- d is a byte array containing the default value.

- C is an array of varying lengths of strings containing the documentation.

Image 14. Pallet Storage



Definition 156. Storage Entry Modifier

> **INFO**
> This might be incorrect and has to be reviewed.

The storage entry modifier is a varying datatype (Definition 178) and indicates how the storage entry is returned and how it behaves if the entry is not present.

$$m = \begin{cases} 0 & \text{optional} \\ 1 & \text{default} \end{cases}$$

where *0* indicates that the entry returns an *Option* type and therefore *None* if the storage entry is not present. *1* indicates that the entry returns the type y with default value d (in Definition 155) if the entry is not present.

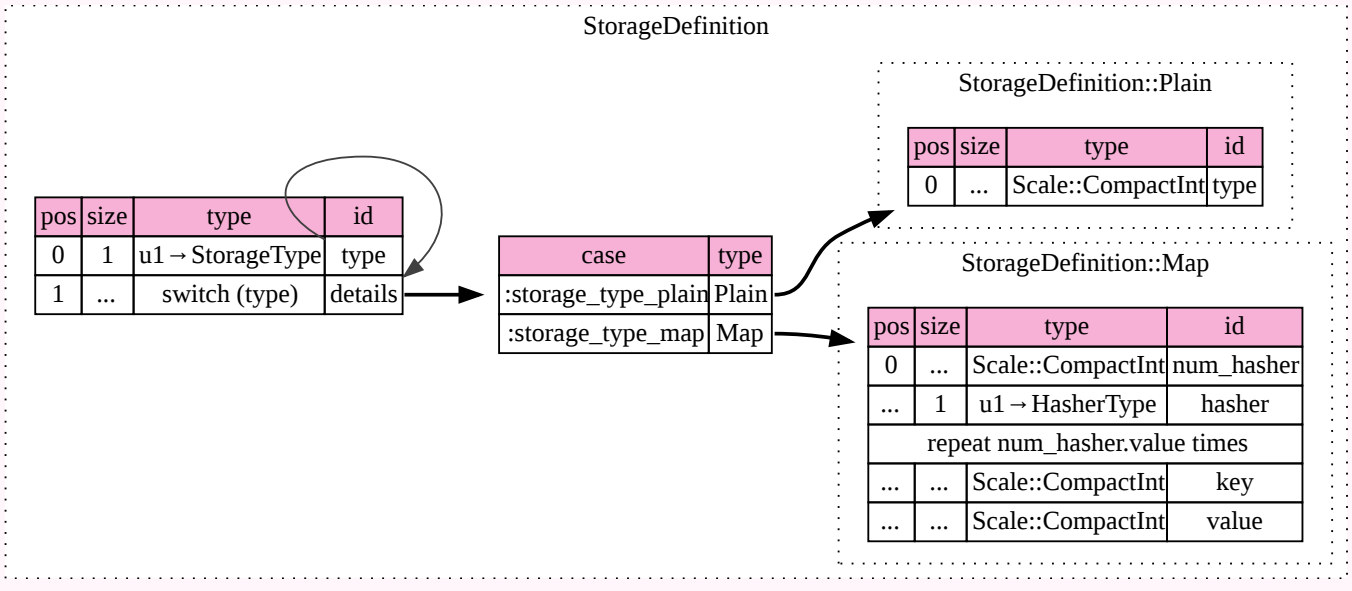
Definition 157. Storage Entry Type

The type of the storage value is a varying datatype (Definition 178) that indicates how the entry is stored.

$$y = \begin{cases} 0 & \rightarrow & t & \text{plain type} \\ 1 & \rightarrow & (H, k, v) & \text{storage map} \end{cases}$$

where t, k (key) and v (value) are all of type Ids (Definition 150). H is an array of varying length containing the storage hasher (Definition 158).

Image 15. Storage Definition



Definition 158. Storage Hasher

The hashing algorithm is used by storage maps.

$$\begin{cases} 0 & \text{128-bit Blake2 hash} \\ 1 & \text{256-bit Blake2 hash} \\ 2 & \text{Multiple 128-bit Blake2 hashes concatenated} \\ 3 & \text{128-bit XX hash} \\ 4 & \text{256-bit XX hash} \\ 5 & \text{Multiple 64-bit XX hashes concatenated} \\ 6 & \text{Identity hashing} \end{cases}$$

Definition 159. Pallet Constants

The metadata about the pallets constants.

$$c_i = (n, y, v, C)$$

where

- n is a string representing the name of the pallet constant.
- y is the type Id (Definition 150) of the pallet constant.
- v is a byte array containing the value of the constant.
- C is an array of varying lengths containing a string with the documentation.

Image 16. Pallet Constant

PalletConstant

pos	size	type	id
0	...	Scale::String	name
...	...	Scale::CompactInt	type
...	...	Scale::Bytes	value
...	...	Scale::StringList	docs

12.3. Extrinsic Metadata

The metadata about a pallets extrinsics, part of the main structure (Section 12.1.) and of the following format:

Definition 160. Signed Extension Metadata

The metadata about the additional, signed data required to execute an extrinsic.

$$e_i = (n, y, a)$$

where

- n is a string representing the unique signed extension identifier, which may be different from the type name.
- y is a type Id (Definition 150) of the signed extension, with the data to be included in the extrinsic.
- a is the type Id (Definition 150) of the additional signed data, with the data to be included in the signed payload.

Image 17. Metadata Extrinsic

MetadataExtrinsic

pos	size	type	id
0	...	Scale::String	name
...	...	Scale::CompactInt	type
...	...	Scale::CompactInt	additional

Appendix A: Cryptography & Encoding

The appendix chapter contains various protocol details.

A.1. Cryptographic Algorithms

A.1.1. Hash Functions

A.1.1.1. 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).

A.1.2. Randomness

> **INFO**
> TBH

A.1.3. 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 Block-Production-Lottery and the parachain approval voting mechanism (Section 8.5.). The VRF uses a mechanism similar to algorithms introduced in the following papers:

- Making NSEC5 Practical for DNSSEC (2)
- DLEQ Proofs
- Verifiable Random Functions (VRFs) (3)

It essentially generates a deterministic elliptic curve based on Schnorr signature as a verifiable random value. The elliptic curve group used in the VRF function is the Ristretto group specified in:

- ristretto.group/

Definition 161. VRF Proof

The **VRF proof** proves the correctness of an associated VRF output. The VRF proof, P, is a data structure of the following format:

$$P = (C, S)$$

$$S = (b_0, \dots b_{31})$$

where C is the challenge and S is the 32-byte Schnorr poof. Both are expressed as Curve25519 scalars as defined in Definition Definition 162.

Definition 162. `DLEQ Prove`

The $\mathrm{dleq\_prove}(t, i)$ function creates a proof for a given input, i, based on the provided transcript, T.

First:

$$t_1 = \mathrm{append}(t, \text{'proto-name'}, \text{'DLEQProof'})$$

$$t_2 = \mathrm{append}(t_1, \text{'vrf:h'}, i)$$

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.

$$\begin{aligned} t_3 &= \text{meta-AD}(t_2, \text{'proving00'}, \text{more=False}) \\ t_4 &= \text{meta-AD}(t_3, w_l, \text{more=True}) \\ t_5 &= \mathrm{KEY}(t_4, w, \text{more=False}) \\ t_6 &= \text{meta-AD}(t_5, \text{'rng'}, \text{more=False}) \\ t_7 &= \mathrm{KEY}(t_6, r, \text{more=False}) \\ t_8 &= \text{meta-AD}(t_7, e\_(64), \text{more=False}) \\ (\phi, s_w) &= \mathrm{PRF}(t_8, \text{more=False}) \end{aligned}$$

where w_l is the length of the witness, encoded as a 32-bit little-endian integer. r is a 32-byte array containing the secret witness scalar.

$$\begin{aligned} l_1 &= \mathrm{append}(t_2, \text{'vrf:R=g}^{r}\text{'}, s_w) \\ l_2 &= \mathrm{append}(l_1, \text{'vrf:h}^{r}\text{'}, s_i) \\ l_3 &= \mathrm{append}(l_2, \text{'vrf:pk'}, s_p) \\ l_4 &= \mathrm{append}(l_3, \text{'vrf:h}^{sk}\text{'}, \mathrm{vrf}_o) \end{aligned}$$

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:

$$\begin{aligned} l_5 &= \text{meta-AD}(l_4, \text{'prove'}, \text{more=False}) \\ l_6 &= \text{meta-AD}(l_5, e_{64}, \text{more=True}) \\ C &= \mathrm{PRF}(l_6, \text{more=False}) \end{aligned}$$

And the Schnorr proof:

$$S = s_w - (C \cdot p)$$

where p is the secret key.

Definition 163. `DLEQ Verify`

The $\mathrm{dleq\_verify}(i, o, P, p_k)$ function verifiers the VRF input, i against the output, o, with the associated proof (Definition 161) and public key, p_k .

$$\begin{aligned} t_1 &= \mathrm{append}(t, \text{'proto-name'}, \text{'DLEQProof'}) \\ t_2 &= \mathrm{append}(t_1, \text{'vrf:h'}, s_i) \\ t_3 &= \mathrm{append}(t_2, \text{'vrf:R=g}^{r}\text{'}, R) \\ t_4 &= \mathrm{append}(t_3, \text{'vrf:h}^{r}\text{'}, H) \\ t_5 &= \mathrm{append}(t_4, \text{'vrf:pk'}, p_k) \\ t_6 &= \mathrm{append}(t_5, \text{'vrf:h}^{sk}\text{'}, o) \end{aligned}$$

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:

$$t_7 = \text{meta-AD}(t_6, \text{'prove'}, \text{more=False})$$

$$t_8 = \text{meta-AD}(t_7, e_{64}, \text{more=True})$$

$$y = \text{PRF}(t_8, \text{more=False})$$

A.1.3.1. 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 8), and is therefore described in more detail in those protocols. The input value used to initiate the transcript is referred to as a *context* (Definition 164).

Definition 164. VRF Context

The **VRF context** is a constant byte array used to initiate the VRF transcript. The VRF context is constant for all users of the VRF for the specific context for which the VRF function is used. Context prevents VRF values generated by the same nodes for other purposes to be reused for purposes not meant to. For example, the VRF context for the BABE Production lottery defined in Section 5.2. is set to be "substrate-babe-vrf".

Definition 165. 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 166) (where `more=False`) is used to add the protocol label `Merlin v1.0` to obj followed by *appending* (Section A.1.3.1.1.) 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 166) 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 166. 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"

A.1.3.1.1. Messages

Appending messages, or "data", to the transcript (Definition 165) 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 165), 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 166.

Formally, when appending a message, we refer to it as $\text{append}(T, l, m)$.

A.1.4. 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 in this Section.

Definition 167. 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 ed25519 signature complies with (4) except for the verification process which adhere to Ed25519 Zebra variant specified in (5). In short, the signature point is not assumed to be in the prime-ordered subgroup group. As such, the verifier must explicitly clear the cofactor during the course of verifying the signature equation.
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 Definition 167, Definition 168 and Definition 169 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 Section A.1.4.2. and Section A.1.4.5. respectively.

Definition 168. Stash Key

The **Stash key** is a type of account key that holds funds bonded for staking (described in Section A.1.4.1.) to a particular controller key (defined in Definition 169). 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 Section A.1.4.2.. 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 169. 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 Section A.1.4.4.. If used for a validator, it certifies the session keys, as described in Section A.1.4.5.. It only needs the required funds to pay transaction fees [TODO: key needing fund needs to be defined].

Definition 170. 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 3:

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 B. Session keys are *not* meant to control the majority of the users' funds and should only be used for their intended purpose.

A.1.4.1. Holding and staking funds

INFO
TBH

A.1.4.2. Creating a Controller key

INFO
TBH

A.1.4.3. Designating a proxy for voting

INFO
TBH

A.1.4.4. Controller settings

INFO
TBH

A.1.4.5. 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 into 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.]

A.2. Auxiliary Encodings

Definition 171. 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.

A.2.1. Binary Enconding

Definition 172. Sequence of Bytes

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

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

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

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

Definition 173. Bitwise Representation

For a given byte $0 \leq b \leq 255$ the **bitwise representation** in bits $b_i \in \{0, 1\}$ is defined as:

$$b = b_7 \ldots b_0$$

where

$$b = 2^7 b_7 + 2^6 b_6 + \ldots + 2^0 b_0$$

Definition 174. Little Endian

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

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

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

$$b_i = B_i$$

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

$$\mathrm{Enc}_{\mathrm{LE}} : \mathbb{Z}^+ \to \mathbb{B}; (B_n \ldots B_0)_{256}| \to (B_{0,} B_1, \ldots, B_n)$$

Definition 175. UINT32

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

A.2.2. 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 176. 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 data structure can be ignored.

> **⚠ CAUTION**
> 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 size, such as sequences (Definition 182). 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 177. Tuple

The **SCALE codec** for **Tuple**, T, such that:

$$T = (A_1, \dots A_n)$$

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

$$\text{Enc}_{\text{SC}}(T) = \text{Enc}_{\text{SC}}(A_1)||\text{Enc}_{\text{SC}}(A_2)||\dots||\text{Enc}_{\text{SC}}(A_n)$$

In the 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 178. Varying Data Type

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

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

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 180) and *Result* (Definition 181).

Definition 179. Encoding of Varying Data Type

The SCALE codec for value $A = (A_{\text{Type}}, A_{\text{Value}})$ of varying data type $\mathcal{T} = \{T_i, \dots T_n\}$, 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 a 8-bit integer determining the type of A. In particular, for the optional type defined in Definition 178, 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 180. 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 181. 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 182. Sequence

The **SCALE codec** for **sequence** S such that:

$$S = A_1, \dots A_n$$

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:

$$\text{Enc}_{\text{SC}}(S) = \text{Enc}_{\text{SC}}^{\text{Len}}(|S|)||\text{Enc}_{\text{SC}}(A_2)|| \dots ||\text{Enc}_{\text{SC}}(A_n)$$

where $\text{Enc}_{\text{SC}}^{\text{Len}}$ is defined in Definition 188.

In some cases, the length indicator $\text{Enc}_{\text{SC}}^{\text{Len}}(|S|)$ 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 183. Dictionary

SCALE codec for **dictionary** or **hashtable** D with key-value pairs (k_i, v_i)s such that:

$$D = \{(k_1, v_1), \dots (k_n, v_n)\}$$

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

$$\text{Enc}_{\text{SC}}(D) = \text{Enc}_{\text{SC}}^{\text{Size}}(|D|)||\text{Enc}_{\text{SC}}(k_1, v_1)|| \dots ||\text{Enc}_{\text{SC}}(k_n, v_n)$$

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 184. Boolean

The SCALE codec for a **boolean value** b defined as a byte as follows:

$$\text{Enc}_{\text{SC}} : \{\text{False}, \text{True}\} \to \mathbb{B}_1$$

$$b \to \begin{cases} 0 & b = \text{False} \\ 1 & b = \text{True} \end{cases}$$

Definition 185. String

The SCALE codec for a **string value** is an encoded sequence (Definition 182) consisting of UTF-8 encoded bytes.

Definition 186. Fixed Length

The SCALE codec, $\mathrm{Enc}_{\mathrm{SC}}$, for other types such as fixed length integers not defined here otherwise, is equal to little endian encoding of those values defined in Definition 174.

Definition 187. Empty

The SCALE codec, $\mathrm{Enc}_{\mathrm{SC}}$, for an empty type is defined to a byte array of zero length and depicted as ϕ.

A.2.2.1. Length and Compact Encoding

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

Definition 188. 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:

$$\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}} : \mathbb{N} \rightarrow \mathbb{B}$$

$$n \rightarrow b = \begin{cases} l_1 & 0 \leq n < 2^6 \\ i_1 i_2 & 2^6 \leq n < 2^{14} \\ j_1 j_2 j_3 j_4 & 2^{14} \leq n < 2^{30} \\ k_1 k_2 \dots k_{m+1} & 2^{30} \leq n \end{cases}$$

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:

$$n = \begin{cases} l_1^7 \dots l_1^3 l_1^2 & n < 2^6 \\ i_2^7 \dots i_2^0 i_1^7 .. i_1^2 & 2^6 \leq n < 2^{14} \\ j_4^7 \dots j_4^0 j_3^7 \dots j_1^7 \dots j_1^2 & 2^{14} \leq n < 2^{30} \\ k_2 + k_3 2^8 + k_4 2^{2 \times 8} + \dots + k_{m+1} 2^{(m-1)8} & 2^{30} \leq n \end{cases}$$

such that:

$$k_1^7 \dots k_1^3 k_1^2 = m - 4$$

Note that m denotes the length of the original integer being encoded and does not include the extra-byte describing the length. The encoding can be used for integers up to $2^{(63+4)8} - 1 = 2^{536} - 1$.

A.2.3. 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 189. Hex Encoding

Suppose that $\mathrm{PK} = (k_1, \dots k_n)$ is a sequence of nibbles, then:

$$\text{Enc}_{\text{HE}}(\text{PK}) = \begin{cases} \text{Nibbles}_4 & \rightarrow & \mathbb{B} \\ \text{PK} = (k_1, \dots k_n) & \rightarrow & \begin{cases} (16k_1 + k_2, \dots, 16k_{2i-1} + k_{2i}) & n = 2i \\ (k_1, 16k_2 + k_3, \dots, 16k_{2i} + k_{2i+1}) & n = 2i + 1 \end{cases} \end{cases}$$

A.3. Chain Specification

Chain Specification (chainspec) is a collection of information that describes the blockchain network. It includes information required for a host to connect and sync with the Polakdot network, for example, the initial nodes to communicate with, protocol identifier, initial state that the hosts agree, etc. There are a set of core fields required by the Host and a set of extensions which are used by optionally implemented features of the Host. The fields of chain specification are categorised in three parts:

1. ChainSpec
2. ChainSpec Extensions
3. Genesis State which is the only mandatory part of the chainspec.

A.3.1. Chain Spec

Chain specification contains information used by the Host to communicate with network participants and optionally send data to telemetry endpoints.

The **client specification** contains the fields below. The values for Polkadot chain are specified:

- *name*: The human readable name of the chain.

```
"name": "Polkadot"
```

- *id*: The id of the chain.

```
"id": "polkadot"
```

- *chainType*: Possible values are `Live`, `Development`, `Local`.

```
"chainType": "Live"
```

- *bootNodes*: A list of MultiAddress that belong to boot nodes of the chain. The list of boot nodes for Polkadot can be found here
- *telemetryEndpoints*: Optional list of "(*multiaddress*, *verbosity*)" pairs of telemetry endpoints. The verbosity goes from `0` to `9`. With `0` being the mode with the lowest verbosity.
- *forkId*: Optional fork id. Should most likely be left empty. Can be used to signal a fork on the network level when two chains have the same genesis hash.

```
"forkId": {}
```

- *properties*: Optional additional properties of the chain as subfields including token symbol, token decimals and address formats.

```
"properties": {
  "ss58Format": 0,
  "tokenDecimals": 10,
  "tokenSymbol": "DOT"
}
```

A.3.2. Chain Spec Extensions

ChainSpec Extensions are additional parameters customisable from the chainspec and correspond to optional features implemented in the Host.

Definition 190. Bad Blocks Header

BadBlocks describes a list of block header hashes that are known apriori to be bad (not belonging to canonical chain) by the host, so that the host can explicitly avoid importing them. These block headers are always considered invalid and filtered out before importing the block:

$$badBlocks = (b_0, \dots b_n)$$

where b_i is a known invalid block header hash.

Definition 191. Fork Blocks

ForkBlocks describes a list of expected block header hashes at certain block heights. They are used to set trusted checkpoints, i.e., the host will refuse to import a block with a different hash at the given height. Forkblocks are useful mechanism to guide the Host to the right fork in instances where the chain is bricked (possibly due to issues in runtime upgrades).

$$forkBlocks = (< b_0, H_0 >, \dots < b_n, H_n >)$$

where b_i is an apriori known valid block header hash at block height H_i. The host is expected to accept no other block except b_i at height H_i.

INFO

lightSyncState describes a check-pointing format for light clients. Its specification is currently Work-In-Progress.

A.3.3. 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 2.6.2.) and `:heappages` (Section 2.6.3.1.) 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 B.2.1.).

As such, Polkadot does not define a formal genesis block. Nonetheless for the compatibility reasons in several algorithms, the Polkadot Host defines the *genesis header* (Definition 192). By the abuse of terminology, "genesis block" refers to the hypothetical parent of block number *1* which holds genesis header as its header.

Definition 192. Genesis Header

The Polkadot genesis header is a data structure conforming to block header format (Definition 10). It contains the following values:

Table 4. Table of Genesis Header Values

Block header field	Genesis Header Value
`parent_hash`	*0*
`number`	*0*
`state_root`	Merkle hash of the state storage trie (Definition 29) after inserting the genesis state in it.
`extrinsics_root`	*0*
`digest`	*0*

Definition 193. Code Substitutes

Code Substitutes is a list of pairs of block number and `wasm_code`. The given WASM code will be used to substitute the on-chain wasm code starting with the given block number until the `spec_version` on-chain changes. The substitute code should be as close as possible to the on-chain wasm code. A substitute should be used to fix a bug that can not be fixed with a runtime upgrade, if for example the runtime is constantly panicking. Introducing new runtime apis isn't supported, because the node will read the runtime version from the on-chain wasm code. Use this functionality only when there is no other way around and to only patch the problematic bug, the rest should be done with a on-chain runtime upgrade.

A.4. Erasure Encoding

A.4.1. Erasure Encoding

INFO

Erasure Encoding has not been documented yet.

Bibliography

1. Saarinen MJ, Aumasson J-P. The BLAKE2 cryptographic hash and message authentication code (MAC) [Internet]. https://tools.ietf.org/html/rfc7693: -; 2015. Report No.: 7693. Available from: https://tools.ietf.org/html/rfc7693
2. Papadopoulos D, Wessels D, Huque S, Naor M, Včelák J, Reyzin L, et al. Making NSEC5 Practical for DNSSEC [Internet]. Cryptology ePrint Archive, Paper 2017/099; 2017. Available from: https://eprint.iacr.org/2017/099
3. Goldberg S, Papadopoulos D, Vcelak J. Internet Draft - Verifiable Random Functions (VRFs) [Internet]. draft-goldbe-vrf-01. 2017. Available from: https://tools.ietf.org/id/draft-goldbe-vrf-01.html
4. Josefsson S, Liusvaara I. Edwards-curve digital signature algorithm (EdDSA). In: Internet Research Task Force, Crypto Forum Research Group, RFC. 2017.
5. de Valence H. Explicitly Defining and Modifying Ed25519 Validation Rules [Internet]. 2020. Available from: https://github.com/zcash/zips/blob/master/zip-0215.rst

Appendix B: Host API

Description of the expected environment available for import by the Polkadot Runtime

B.1. 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 194. Exposed Host API

By RE_B we refer to the API exposed by the Polkadot Host, which interacts, manipulates, and responds based on the state storage whose state is set at the end of the execution of block B.

Definition 195. 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 196. 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 195). 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 197. Lexicographic ordering

Lexicographic ordering refers to the ascending ordering of bytes or byte arrays, such as:

$$[0,0,2] < [0,1,1] < [1] < [1,1,0] < [2] < [\ldots]$$

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

B.2. Storage

Interface for accessing the storage from within the runtime.

DANGER

As of now, the storage API should silently ignore any keys that start with the `:child_storage:default:` prefix. This applies to reading and writing. If the function expects a return value, then *None* (Definition 180) should be returned. See substrate issue #12461.

Definition 198. State Version

The state version, v, dictates how a Merkle root should be constructed. The data structure is a varying type of the following format:

$$v = \begin{cases} 0 & \text{full values} \\ 1 & \text{node hashes} \end{cases}$$

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 28).

B.2.1. `ext_storage_set`

Sets the value under a given key into storage.

B.2.1.1. Version 1 - Prototype

```
(func $ext_storage_set_version_1
    (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `value`: a pointer-size (Definition 196) containing the value.

B.2.2. `ext_storage_get`

Retrieves the value associated with the given key from storage.

B.2.2.1. Version 1 - Prototype

```
(func $ext_storage_get_version_1
    (param $key i64) (result i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `result`: a pointer-size (Definition 196) returning the SCALE encoded *Option* value (Definition 180) containing the value.

B.2.3. `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.

B.2.3.1. Version 1 - Prototype

```
(func $ext_storage_read_version_1
    (param $key i64) (param $value_out i64) (param $offset i32) (result i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `value_out`: a pointer-size (Definition 196) 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 (typed as i32 due to wasm types) containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition 196) pointing to a SCALE encoded *Option* value (Definition 180) containing an unsigned 32-bit integer representing the number of bytes left at supplied `offset`. Returns *None* if the entry does not exist.

B.2.4. `ext_storage_clear`

Clears the storage of the given key and its value. Non-existent entries are silently ignored.

B.2.4.1. Version 1 - Prototype

```
(func $ext_storage_clear_version_1
    (param $key_data i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.

B.2.5. `ext_storage_exists`

Checks whether the given key exists in storage.

B.2.5.1. Version 1 - Prototype

```
(func $ext_storage_exists_version_1
    (param $key_data i64) (return i32))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `return`: an i32 integer value equal to *1* if the key exists or a value equal to *0* if otherwise.

B.2.6. `ext_storage_clear_prefix`

Clear the storage of each key/value pair where the key starts with the given prefix.

B.2.6.1. Version 1 - Prototype

```
(func $ext_storage_clear_prefix_version_1
    (param $prefix i64))
```

Arguments

- `prefix`: a pointer-size (Definition 196) containing the prefix.

B.2.6.2. Version 2 - Prototype

```
(func $ext_storage_clear_prefix_version_2
    (param $prefix i64) (param $limit i64)
    (return i64))
```

Arguments

- `prefix`: a pointer-size (Definition 196) containing the prefix.
- `limit`: a pointer-size (Definition 196) to an *Option* type (Definition 180) 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 toward the limit.
- `return`: a pointer-size (Definition 196) to the following variant, k:

$$k = \begin{cases} 0 & \to c \\ 1 & \to c \end{cases}$$

  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.

B.2.7. `ext_storage_append`

Append the SCALE encoded value to a SCALE encoded sequence (Definition 182) 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 $\mathrm{Enc}_{\mathrm{SC}}^{\mathrm{Len}}$.

> **CAUTION**
> 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.

B.2.7.1. Version 1 - Prototype

```
(func $ext_storage_append_version_1
    (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `value`: a pointer-size (Definition 196) containing the value to be appended.

B.2.8. `ext_storage_root`

Compute the storage root.

B.2.8.1. Version 1 - Prototype

```
(func $ext_storage_root_version_1
    (return i64))
```

Arguments

- `return`: a pointer-size (Definition 196) to a buffer containing the 256-bit Blake2 storage root.

B.2.8.2. Version 2 - Prototype

```
(func $ext_storage_root_version_2
    (param $version i32) (return i64))
```

Arguments

- `version`: the state version (Definition 198).
- `return`: a pointer-size (Definition 196) to the buffer containing the 256-bit Blake2 storage root.

B.2.9. `ext_storage_changes_root`

> **INFO**
> This function is not longer used and only exists for compatibility reasons.

B.2.9.1. Version 1 - Prototype

```
(func $ext_storage_changes_root_version_1
    (param $parent_hash i64) (return i64))
```

Arguments

- `parent_hash`: a pointer-size (Definition 196) to the SCALE encoded block hash.
- `return`: a pointer-size (Definition 196) to an *Option* type (Definition 180) that's always *None*.

B.2.10. `ext_storage_next_key`

Get the next key in storage after the given one in lexicographic order (Definition 197). The key provided to this function may or may not exist in storage.

B.2.10.1. Version 1 - Prototype

```
(func $ext_storage_next_key_version_1
    (param $key i64) (return i64))
```

Arguments

- `key`: a pointer-size (Definition 196) to the key.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the next key in lexicographic order.

B.2.11. `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 B.2.12.) or `ext_storage_commit_transaction` (Section B.2.13.). This is also effective for all values manipulated using the child storage API (Section B.3.). It's legal to call this function multiple times in a row.

> **⚠ CAUTION**
> 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.

B.2.11.1. Version 1 - Prototype

```
(func $ext_storage_start_transaction_version_1)
```

Arguments

- None.

B.2.12. `ext_storage_rollback_transaction`

Rollback the last transaction started by `ext_storage_start_transaction` (Section B.2.11.). Any changes made during that transaction are discarded. It's legal to call this function multiple times in a row.

> **⚠ CAUTION**
> Panics if `ext_storage_start_transaction` (Section B.2.11.) was not called.

B.2.12.1. Version 1 - Prototype

```
(func $ext_storage_rollback_transaction_version_1)
```

Arguments

- None.

B.2.13. `ext_storage_commit_transaction`

Commit the last transaction started by `ext_storage_start_transaction` (Section B.2.11.). Any changes made during that transaction are committed to the main state. It's legal to call this function multiple times in a row.

> **CAUTION**
> Panics if `ext_storage_start_transaction` (Section B.2.11.) was not called.

B.2.13.1. Version 1 - Prototype

```
(func $ext_storage_commit_transaction_version_1)
```

Arguments

- None.

B.3. Child Storage

Interface for accessing the child storage from within the runtime.

Definition 199. Child Storage

> **Child storage** key is an unprefixed location of the child trie in the main trie.

B.3.1. `ext_default_child_storage_set`

Sets the value under a given key into the child storage.

B.3.1.1. 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 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.

B.3.2. `ext_default_child_storage_get`

Retrieves the value associated with the given key from the child storage.

B.3.2.1. 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 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the value.

B.3.3. `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.

B.3.3.1. 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 i32) (result i64))
```

Arguments

- `child_storage_key`: a pointer-size (Definition 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.
- `value_out`: a pointer-size (Definition 196) 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 (typed as i32 due to wasm types) containing the offset beyond the value should be read from.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the number of bytes written into the **value_out** buffer. Returns if the entry does not exists.

B.3.4. `ext_default_child_storage_clear`

Clears the storage of the given key and its value from the child storage. Non-existent entries are silently ignored.

B.3.4.1. 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 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.

B.3.5. `ext_default_child_storage_storage_kill`

Clears an entire child storage.

B.3.5.1. 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 196) to the child storage key (Definition 199).

B.3.5.2. 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 196) to the child storage key (Definition 199).

- `limit`: a pointer-size (Definition 196) to an *Option* type (Definition 180) 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 toward 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.

B.3.5.3. 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 196) to the child storage key (Definition 199).
- `limit`: a pointer-size (Definition 196) to an *Option* type (Definition 180) 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 toward the limit.
- `return`: a pointer-size (Definition 196) to the following variant, k:

$$k = \begin{cases} 0 & \to c \\ 1 & \to c \end{cases}$$

  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.

B.3.6. `ext_default_child_storage_exists`

Checks whether the given key exists in the child storage.

B.3.6.1. 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 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.
- `return`: an i32 integer value equal to *1* if the key exists or a value equal to *0* if otherwise.

B.3.7. `ext_default_child_storage_clear_prefix`

Clears the child storage of each key/value pair where the key starts with the given prefix.

B.3.7.1. 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 196) to the child storage key (Definition 199).
- `prefix`: a pointer-size (Definition 196) to the prefix.

B.3.7.2. 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 196) to the child storage key (Definition 199).
- `prefix`: a pointer-size (Definition 196) to the prefix.
- `limit`: a pointer-size (Definition 196) to an *Option* type (Definition 180) 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 196) to the following variant, k:

$$k = \begin{cases} 0 & \to c \\ 1 & \to c \end{cases}$$

  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.

B.3.8. `ext_default_child_storage_root`

Commits all existing operations and computes the resulting child storage root.

B.3.8.1. 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 196) to the child storage key (Definition 199).
- `return`: a pointer-size (Definition 196) to the SCALE encoded storage root.

B.3.8.2. 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 196) to the child storage key (Definition 199).
- `version`: the state version (Definition 198).
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit Blake2 storage root.

B.3.9. `ext_default_child_storage_next_key`

Gets the next key in storage after the given one in lexicographic order (Definition 197). The key provided to this function may or may not exist in storage.

B.3.9.1. 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 196) to the child storage key (Definition 199).
- `key`: a pointer-size (Definition 196) to the key.
- `return`: a pointer-size (Definition 196) to the SCALE encoded as defined in Definition 180 containing the next key in lexicographic order. Returns if the entry cannot be found.

B.4. Crypto

Interfaces for working with crypto related types from within the runtime.

Definition 200. 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 5. 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 201. ECDSA Verify Error

EcdsaVerifyError is a varying data type (Definition 178) that specifies the error type when using ECDSA recovery functionality. The following values are possible:

Table 6. Table of error types in ECDSA recovery

Id	Description
0	Incorrect value of R or S
1	Incorrect value of V
2	Invalid signature

B.4.1. `ext_crypto_ed25519_public_keys`

Returns all *ed25519* public keys for the given key identifier from the keystore.

B.4.1.1. 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 195) to the key type identifier (Definition 200).
- `return`: a pointer-size (Definition 196) to an SCALE encoded 256-bit public keys.

B.4.2. `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.

> **CAUTION**
> Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

B.4.2.1. 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 195) to the key type identifier (Definition 200).
- `seed`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the BIP-39 seed which must be valid UTF8.
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit public key.

B.4.3. `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.

B.4.3.1. 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 195) to the key type identifier (Definition 200).
- `key`: a pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size (Definition 196) to the message that is to be signed.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the 64-byte signature. This function returns if the public key cannot be found in the key store.

B.4.4. `ext_crypto_ed25519_verify`

Verifies an *ed25519* signature.

B.4.4.1. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) 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.

B.4.5. `ext_crypto_ed25519_batch_verify`

Registers an ed25519 signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section B.4.20.). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section B.4.21.). If batch verification is not enabled, the signature is verified immediately.

B.4.5.1. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: an i32 integer value equal to *1* if the signature is valid or batched or a value equal *0* to if otherwise.

B.4.6. `ext_crypto_sr25519_public_keys`

Returns all *sr25519* public keys for the given key id from the keystore.

B.4.6.1. 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 195) to the key type identifier (Definition 200).
- `return`: a pointer-size (Definition 196) to the SCALE encoded 256-bit public keys.

B.4.7. `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.

> **CAUTION**
> Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

B.4.7.1. 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 195) to the key identifier (Definition 200).
- `seed`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the BIP-39 seed which must be valid UTF8.

- `return`: a pointer (Definition 195) to the buffer containing the 256-bit public key.

B.4.8. `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.

B.4.8.1. 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 195) to the key identifier (Definition 200).
- `key`: a pointer to the buffer containing the 256-bit public key.
- `msg`: a pointer-size (Definition 196) to the message that is to be signed.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the 64-byte signature. This function returns *None* if the public key cannot be found in the key store.

B.4.9. `ext_crypto_sr25519_verify`

Verifies an sr25519 signature.

B.4.9.1. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) 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.

B.4.9.2. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) 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.

B.4.10. `ext_crypto_sr25519_batch_verify`

Registers a sr25519 signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section B.4.20.). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section B.4.21.). If batch verification is not enabled, the signature is verified immediately.

B.4.10.1. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) to the message that is to be verified.
- `key`: a pointer to the buffer containing the 256-bit public key.
- `return`: an i32 integer value equal to *1* if the signature is valid or batched or a value equal *0* to if otherwise.

B.4.11. `ext_crypto_ecdsa_public_keys`

Returns all *ecdsa* public keys for the given key id from the keystore.

B.4.11.1. 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 195) to the key type identifier (Definition 200).
- `return`: a pointer-size (Definition 196) to the SCALE encoded 33-byte compressed public keys.

B.4.12. `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.

> **CAUTION**
> Panics if the key cannot be generated, such as when an invalid key type or invalid seed was provided.

B.4.12.1. 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 195) to the key identifier (Definition 200).
- `seed`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the BIP-39 seed which must be valid UTF8.
- `return`: a pointer (Definition 195) to the buffer containing the 33-byte compressed public key.

B.4.13. `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.

B.4.13.1. 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 195) to the key identifier (Definition 200).

- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size (Definition 196) to the message that is to be signed.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) 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.

B.4.14. `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.

B.4.14.1. 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 195) to the key identifier (Definition 200).
- `key`: a pointer to the buffer containing the 33-byte compressed public key.
- `msg`: a pointer-size (Definition 196) to the message that is to be signed.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) 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.

B.4.15. `ext_crypto_ecdsa_verify`

Verifies the hash of the given message against an ECDSA signature.

B.4.15.1. Version 1 - Prototype

This function allows the verification of non-standard, overflowing ECDSA signatures, an implementation 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 195) 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 196) 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.

B.4.15.2. 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 195) 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 196) 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.

B.4.16. `ext_crypto_ecdsa_verify_prehashed`

Verifies the prehashed message against a ECDSA signature.

B.4.16.1. 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 195) 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.

B.4.17. `ext_crypto_ecdsa_batch_verify`

Registers a ECDSA signature for batch verification. Batch verification is enabled by calling `ext_crypto_start_batch_verify` (Section B.4.20.). The result of the verification is returned by `ext_crypto_finish_batch_verify` (Section B.4.21.). If batch verification is not enabled, the signature is verified immediately.

B.4.17.1. 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 195) to the buffer containing the 64-byte signature.
- `msg`: a pointer-size (Definition 196) 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.

B.4.18. `ext_crypto_secp256k1_ecdsa_recover`

Verify and recover a *secp256k1* ECDSA signature.

B.4.18.1. 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 195) to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.

- `msg`: a pointer (Definition 195) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Result* (Definition 181). On success it contains the 64-byte recovered public key or an error type (Definition 201) on failure.

B.4.18.2. 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 195) to the buffer containing the 65-byte signature in RSV format. V should be either or .
- `msg`: a pointer (Definition 195) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Result* (Definition 181). On success it contains the 64-byte recovered public key or an error type (Definition 201) on failure.

B.4.19. `ext_crypto_secp256k1_ecdsa_recover_compressed`

Verify and recover a *secp256k1* ECDSA signature.

B.4.19.1. 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 195) to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a pointer (Definition 195) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 196) to the SCALE encoded `Result` value (Definition 181). On success it contains the 33-byte recovered public key in compressed form on success or an error type (Definition 201) on failure.

B.4.19.2. 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 195) to the buffer containing the 65-byte signature in RSV format. V should be either `0/1` or `27/28`.
- `msg`: a pointer (Definition 195) to the buffer containing the 256-bit Blake2 hash of the message.
- `return`: a pointer-size (Definition 196) to the SCALE encoded `Result` value (Definition 181). On success it contains the 33-byte recovered public key in compressed form on success or an error type (Definition 201) on failure.

B.4.20. `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 B.4.5.), `ext_crypto_sr25519_batch_verify` (Section B.4.10.) or `ext_crypto_ecdsa_batch_verify` (Section B.4.17.). Verification will start immediately and the Runtime can retrieve the result when calling

`ext_crypto_finish_batch_verify` (Section B.4.21.).

B.4.20.1. Version 1 - Prototype

```
(func $ext_crypto_start_batch_verify_version_1)
```

Arguments

- None.

B.4.21. `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.

> **CAUTION**
> Panics if `ext_crypto_start_batch_verify` (Section B.4.20.) was not called.

B.4.21.1. 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.

B.5. Hashing

Interface that provides functions for hashing with different algorithms.

B.5.1. `ext_hashing_keccak_256`

Conducts a 256-bit Keccak hash.

B.5.1.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit hash result.

B.5.2. `ext_hashing_keccak_512`

Conducts a 512-bit Keccak hash.

B.5.2.1. Version 1 - Prototype

```
(func $ext_hashing_keccak_512_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.

- `return`: a pointer (Definition 195) to the buffer containing the 512-bit hash result.

B.5.3. `ext_hashing_sha2_256`

Conducts a 256-bit Sha2 hash.

B.5.3.1. Version 1 - Prototype

```
(func $ext_hashing_sha2_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit hash result.

B.5.4. `ext_hashing_blake2_128`

Conducts a 128-bit Blake2 hash.

B.5.4.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_128_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 128-bit hash result.

B.5.5. `ext_hashing_blake2_256`

Conducts a 256-bit Blake2 hash.

B.5.5.1. Version 1 - Prototype

```
(func $ext_hashing_blake2_256_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit hash result.

B.5.6. `ext_hashing_twox_64`

Conducts a 64-bit xxHash hash.

B.5.6.1. Version 1 - Prototype

```
(func $ext_hashing_twox_64_version_1
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 64-bit hash result.

B.5.7. `ext_hashing_twox_128`

Conducts a 128-bit xxHash hash.

B.5.7.1. Version 1 - Prototype

```
(func $ext_hashing_twox_128
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 128-bit hash result.

B.5.8. `ext_hashing_twox_256`

Conducts a 256-bit xxHash hash.

B.5.8.1. Version 1 - Prototype

```
(func $ext_hashing_twox_256
    (param $data i64) (return i32))
```

Arguments

- `data`: a pointer-size (Definition 196) to the data to be hashed.
- `return`: a pointer (Definition 195) to the buffer containing the 256-bit hash result.

B.6. 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 202 and Definition 203.

Definition 202. 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 203. 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 204. 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 205. HTTP Error

HTTP error, E, is a varying data type (Definition 178) and specifies the error types of certain HTTP functions. Following values are possible:

$$E = \begin{cases} 0 & \text{The deadile was reached} \\ 1 & \text{There was an IO error while processing the request} \\ 2 & \text{The Id of the request is invalid} \end{cases}$$

B.6.1. `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.

B.6.1.1. 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.

B.6.2. `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.

B.6.2.1. Version 1 - Prototype

```
(func $ext_offchain_submit_transaction_version_1
    (param $data i64) (return i64))
```

Arguments

- `data`: a pointer-size (Definition 196) to the byte array storing the encoded extrinsic.
- `return`: a pointer-size (Definition 196) to the SCALE encoded *Result* value (Definition 181). Neither on success or failure is there any additional data provided. The cause of a failure is implementation specific.

B.6.3. `ext_offchain_network_state`

Returns the SCALE encoded, opaque information about the local node's network state.

Definition 206. 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*, $(M_0, \dots M_n)$, the node knows it can be reached at:

$$S = (P_{\text{id}}, (M_0, \dots M_n))$$

where

$$P_{\text{id}} = (b_0, \dots b_n)$$

$$M = (b_0, \dots b_n)$$

The information contained in this structure is naturally opaque to the caller of this function.

B.6.3.1. Version 1 - Prototype

```
(func $ext_offchain_network_state_version_1 (result i64))
```

Arguments

- `result`: a pointer-size (Definition 196) to the SCALE encoded `Result` value (Definition 181). On success it contains the *Opaque network state* structure (Definition 206). On failure, an empty value is yielded where its cause is implementation specific.

B.6.4. `ext_offchain_timestamp`

Returns the current timestamp.

B.6.4.1. Version 1 - Prototype

```
(func $ext_offchain_timestamp_version_1 (result i64))
```

Arguments

- `result`: an u64 integer (typed as i64 due to wasm types) indicating the current UNIX timestamp (Definition 171).

B.6.5. `ext_offchain_sleep_until`

Pause the execution until the `deadline` is reached.

B.6.5.1. Version 1 - Prototype

```
(func $ext_offchain_sleep_until_version_1 (param $deadline i64))
```

Arguments

- `deadline`: an u64 integer (typed as i64 due to wasm types) specifying the UNIX timestamp (Definition 171).

B.6.6. `ext_offchain_random_seed`

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

B.6.6.1. Version 1 - Prototype

```
(func $ext_offchain_random_seed_version_1 (result i32))
```

Arguments

- `result`: a pointer (Definition 195) to the buffer containing the 256-bit seed.

B.6.7. `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.

B.6.7.1. 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 202) and a value equal to *2* for local storage (Definition 203).
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.

B.6.8. `ext_offchain_local_storage_clear`

Remove a value from the local storage.

B.6.8.1. 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 202) and a value equal to *2* for local storage (Definition 203).
- `key`: a pointer-size (Definition 196) to the key.

B.6.9. `ext_offchain_local_storage_compare_and_set`

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

B.6.9.1. 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 202) and a value equal to *2* for local storage (Definition 203).
- `key`: a pointer-size (Definition 196) to the key.
- `old_value`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the old key.
- `new_value`: a pointer-size (Definition 196) 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.

B.6.10. `ext_offchain_local_storage_get`

Gets a value from the local storage.

B.6.10.1. 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 202) and a value equal to *2* for local storage (Definition 203).
- `key`: a pointer-size (Definition 196) to the key.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the value or the corresponding key.

B.6.11. `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.

B.6.11.1. 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 196) to the HTTP method. Possible values are “GET” and “POST”.
- `uri`: a pointer-size (Definition 196) 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 196) to the SCALE encoded *Result* value (Definition 181) containing the i16 ID of the newly started request. On failure no additionally data is provided. The cause of failure is implementation specific.

B.6.12. `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).

B.6.12.1. 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 196) to the HTTP header name.
- `value`: a pointer-size (Definition 196) to the HTTP header value.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Result* value (Definition 181). Neither on success or failure is there any additional data provided. The cause of failure is implementation specific.

B.6.13. `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.

B.6.13.1. 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 196) to the chunk of bytes. Writing an empty chunk finalizes the request.

- `deadline`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the UNIX timestamp (Definition 171). Passing *None* blocks indefinitely.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Result* value (Definition 181). On success, no additional data is provided. On error it contains the HTTP error type (Definition 205).

B.6.14. `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.

B.6.14.1. 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 196) to the SCALE encoded array of started request IDs.
- `deadline`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the UNIX timestamp (Definition 171). Passing None blocks indefinitely.
- `result`: a pointer-size (Definition 196) to the SCALE encoded array of request statuses (Definition 204).

B.6.15. `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.

B.6.15.1. 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 196) to a SCALE encoded array of key/value pairs.

B.6.16. `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.

B.6.16.1. 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 196) to the buffer where the body gets written to.
- `deadline`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the UNIX timestamp (Definition 171). Passing *None* will block indefinitely.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Result* value (Definition 181). On success it contains an i32 integer specifying the number of bytes written or a HTTP error type (Definition 205) on failure.

B.7. Offchain Index

Interface that provides functions to access the Offchain DB through offchain indexing.

B.7.1. `Offchain_index_set`

Write a key-value pair to the Offchain DB in a buffered fashion.

B.7.1.1. Version 1 - Prototype

```
(func $ext_offchain_index_set_version_1
    (param $key i64) (param $value i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.
- `value`: a pointer-size (Definition 196) containing the value.

B.7.2. `Offchain_index_clear`

Remove a key and its associated value from the Offchain DB.

B.7.2.1. Version 1 - Prototype

```
(func $ext_offchain_index_set_version_1
    (param $key i64))
```

Arguments

- `key`: a pointer-size (Definition 196) containing the key.

B.8. Trie

Interface that provides trie related functionality.

B.8.1. `ext_trie_blake2_256_root`

Compute a 256-bit Blake2 trie root formed from the iterated items.

B.8.1.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 196) 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 195) to the buffer containing the 256-bit trie root.

B.8.1.2. 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 196) 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`: the state version (Definition 198).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root.

B.8.2. `ext_trie_blake2_256_ordered_root`

Compute a 256-bit Blake2 trie root formed from the enumerated items.

B.8.2.1. Version 1 - Prototype

```
(func $ext_trie_blake2_256_ordered_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 196) 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 188).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root result.

B.8.2.2. 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 196) 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 188).
- `version`: the state version (Definition 198).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root result.

B.8.3. `ext_trie_keccak_256_root`

Compute a 256-bit Keccak trie root formed from the iterated items.

B.8.3.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 196) 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 195) to the buffer containing the 256-bit trie root.

B.8.3.2. 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 196) 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`: the state version (Definition 198).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root.

B.8.4. `ext_trie_keccak_256_ordered_root`

Compute a 256-bit Keccak trie root formed from the enumerated items.

B.8.4.1. Version 1 - Prototype

```
(func $ext_trie_keccak_256_ordered_root_version_1
    (param $data i64) (result i32))
```

Arguments

- `data`: a pointer-size (Definition 196) 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 188).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root result.

B.8.4.2. 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 196) 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 188).
- `version`: the state version (Definition 198).
- `result`: a pointer (Definition 195) to the buffer containing the 256-bit trie root result.

B.8.5. `ext_trie_blake2_256_verify_proof`

Verifies a key/value pair against a Blake2 256-bit merkle root.

B.8.5.1. 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 196) to an array containing the node proofs.
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.

- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

B.8.5.2. 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 196) to an array containing the node proofs.
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.
- `version`: the state version (Definition 198).
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

B.8.6. `ext_trie_keccak_256_verify_proof`

Verifies a key/value pair against a Keccak 256-bit merkle root.

B.8.6.1. 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 196) to an array containing the node proofs.
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.
- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

B.8.6.2. 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 196) to an array containing the node proofs.
- `key`: a pointer-size (Definition 196) to the key.
- `value`: a pointer-size (Definition 196) to the value.
- `version`: the state version (Definition 198).

- `return`: a value equal to *1* if the proof could be successfully verified or a value equal to *0* if otherwise.

B.9. Miscellaneous

Interface that provides miscellaneous functions for communicating between the runtime and the node.

B.9.1. `ext_misc_print_num`

Print a number.

B.9.1.1. Version 1 - Prototype

```
(func $ext_misc_print_num_version_1 (param $value i64))
```

Arguments

- `value`: the number to be printed.

B.9.2. `ext_misc_print_utf8`

Print a valid UTF8 encoded buffer.

B.9.2.1. Version 1 - Prototype

```
(func $ext_misc_print_utf8_version_1 (param $data i64))
```

Arguments:

- : a pointer-size (Definition 196) to the valid buffer to be printed.

B.9.3. `ext_misc_print_hex`

Print any buffer in hexadecimal representation.

B.9.3.1. Version 1 - Prototype

```
(func $ext_misc_print_hex_version_1 (param $data i64))
```

Arguments:

- `data`: a pointer-size (Definition 196) to the buffer to be printed.

B.9.4. `ext_misc_runtime_version`

Extract the Runtime version of the given Wasm blob by calling `Core_version` (Section C.4.1.). Returns the SCALE encoded runtime version or *None* (Definition 180) if the call fails. This function gets primarily used when upgrading Runtimes.

> **⚠ CAUTION**
> 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 2.6.2.) and calling the `Core_version` function (Section C.4.1.) in this blob.

B.9.4.1. Version 1 - Prototype

```
(func $ext_misc_runtime_version_version_1 (param $data i64) (result i64))
```

Arguments

- `data`: a pointer-size (Definition 196) to the Wasm blob.
- `result`: a pointer-size (Definition 196) to the SCALE encoded *Option* value (Definition 180) containing the Runtime version of the given Wasm blob which is encoded as a byte array.

B.10. 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.

B.10.1. `ext_allocator_malloc`

Allocates the given number of bytes and returns the pointer to that memory location.

B.10.1.1. 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 195) to the allocated buffer.

B.10.2. `ext_allocator_free`

Free the given pointer.

B.10.2.1. Version 1 - Prototype

```
(func $ext_allocator_free_version_1 (param $ptr i32))
```

Arguments

- `ptr`: a pointer (Definition 195) to the memory buffer to be freed.

B.11. Logging

Interface that provides functions for logging from within the runtime.

Definition 207. Log Level

The **Log Level**, L, is a varying data type (Definition 178) and implies the emergency of the log. Possible log levels and the corresponding identifier is as follows:

$$L = \begin{cases} 0 & \text{Error} = 1 \\ 1 & \text{Warn} = 2 \\ 2 & \text{Info} = 3 \\ 3 & \text{Debug} = 4 \\ 4 & \text{Trace} = 5 \end{cases}$$

B.11.1. `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.

B.11.1.1. Version 1 - Prototype

```
(func $ext_logging_log_version_1
    (param $level i32) (param $target i64) (param $message i64))
```

Arguments

- `level`: the log level (Definition 207).
- `target`: a pointer-size (Definition 196) to the string which contains the path, module or location from where the log was executed.
- `message`: a pointer-size (Definition 196) to the UTF-8 encoded log message.

B.11.2. `ext_logging_max_level`

Returns the max logging level used by the host.

B.11.2.1. Version 1 - Prototype

```
(func $ext_logging_max_level_version_1
     (result i32))
```

Arguments

- *None*

Returns

- `result`: the max log level (Definition 207) used by the host.

B.12. Abort Handler

Interface for aborting the execution of the runtime.

B.12.1. `ext_panic_handler_abort_on_panic`

Aborts the execution of the runtime with a given message. Note that the message will be only displayed if the host is enabled to display those types of messages, which is implementation specific.

B.12.1.1. Version 1 - Prototype

```
(func $ext_panic_handler_abort_on_panic_version_1
    (param $message i64))
```

Arguments

- `message`: a pointer-size (Definition 196) to the UTF-8 encoded message.

Appendix C: Runtime API

Description of how to interact with the Runtime through its exported functions

C.1. 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 C.4.1.) after every Runtime upgrade and warn if an updated API is encountered and that this might require an update of the host.

This section describes all Runtime API functions alongside their arguments and the return values. The functions are organized into modules, with each being versioned independently.

C.1.1. 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.

C.2. Runtime Constants

C.2.1. `__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 Section 2.6.3.1..

C.3. Runtime Call Convention

Definition 208. 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 196) to the SCALE-encoded return data.

See Section 2.6.3. for more information about the behavior of the Wasm Runtime. Also, note that any storage changes must be fork-aware (Section 2.4.5.).

C.4. Module Core

> **NOTE**
> This section describes **Version 3** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

C.4.1. `Core_version`

> **NOTE**
> For newer Runtimes, the version identifiers can be read directly from the Wasm blob in the form of custom sections (Section 2.6.3.4.). That method of retrieving this data should be preferred since it involves significantly less overhead.

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 C.1.1..

Arguments

- None

Return

- A data structure of the following format:

Table 7. 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
`impl_version`	Unsigned 32-bit integer	Version of the Runtime implementation
`apis`	ApiVersions (Definition 209)	List of supported APIs along with their version
`transaction_version`	Unsigned 32-bit integer	Version of the transaction format
`state_version`	Unsigned 8-bit integer	Version of the trie format

Definition 209. ApiVersions

ApiVersions is a specialized type for the (Section C.4.1.) 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.

$$\begin{aligned}\text{ApiVersions} :=& (T_0, \ldots, T_n) \\ T :=& ((b_0, \ldots, b_7), \text{UINT32})\end{aligned}$$

Requires `Core_initialize_block` to be called beforehand.

C.4.2. `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 Build-Block, 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 resulting 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 11, 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 10, and the block body, as described in Definition 13.

Return

- None.

C.4.3. `Core_initialize_block`

Sets up the environment required for building a new block as described in Build-Block.

Arguments

- The header of the new block as defined in Definition 10. The values H_r, H_e and H_d are left empty.

Return

- None.

C.5. Module Metadata

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

C.5.1. `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 C.1.1., and returns all the information necessary to build valid transactions.

Arguments

- None.

Return

- The scale-encoded (Section A.2.2.) runtime metadata as described in Chapter 12.

C.5.2. `Metadata_metadata_at_version`

Returns native Runtime metadata in an opaque form at a particular version.

Arguments

- Metadata version represented by an unsigned 32-bit integer.

Return

- The scale-encoded (Section A.2.2.) runtime metadata as described in Chapter 12 at the particular version.

C.5.3. `Metadata_metadata_versions`

Returns supported metadata versions.

Arguments

- None.

Return

- A vector of supported metadata versions of type `vec<u32>`.

C.6. Module BlockBuilder

> **NOTE**
> This section describes **Version 4** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.6.1. `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 sizes containing the opaque extrinsic.

Return

- Returns the varying datatype *ApplyExtrinsicResult* as defined in Definition 210. This structure lets the block builder know whether an extrinsic should be included in the block or rejected.

Definition 210. ApplyExtrinsicResult

ApplyExtrinsicResult is a varying data type as defined in Definition 181. This structure can contain multiple nested structures, indicating either module dispatch outcomes or transaction invalidity errors.

Table 8. Possible values of varying data type *ApplyExtrinsicResult*.

Id	Description	Type
0	Outcome of dispatching the extrinsic.	*DispatchOutcome* (Definition 211)
1	Possible errors while checking the validity of a transaction.	*TransactionValidityError* (Definition 214)

> **INFO**
> As long as a *DispatchOutcome* (Definition 211) 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 211. DispatchOutcome

DispatchOutcome is the varying data type as defined in Definition 181.

Table 9. 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 212)

Definition 212. DispatchError

DispatchError is a varying data type as defined in Definition 178. Indicates various reasons why a dispatch call failed.

Table 10. 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 look up some data.	None
2	A bad origin.	None
3	A custom error in a module.	*CustomModuleError* (Definition 213)

Definition 213. CustomModuleError

CustomModuleError is a tuple appended after a possible error in as defined in Definition 212.

Table 11. 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 180). The optional value is a SCALE-encoded byte array containing a valid UTF-8 sequence.

INFO

Whenever *TransactionValidityError* (Definition 214) 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 215 and respectively Definition 216.

Definition 214. TransactionValidityError

TransactionValidityError is a varying data type as defined in Definition 178. It indicates possible errors that can occur while checking the validity of a transaction.

Table 12. Possible values of varying data type *TransactionValidityError*.

Id	Description	Type
0	Transaction is invalid.	*InvalidTransaction* (Definition 215)
1	Transaction validity can't be determined.	*UnknownTransaction* (Definition 216)

Definition 215. InvalidTransaction

InvalidTransaction is a varying data type as defined in Definition 178 and specifies the invalidity of the transaction in more detail.

Table 13. 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

Id	Description	Type	Reject
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 216. UnknownTransaction

UnknownTransaction is a varying data type as defined in Definition 178 and specifies the unknown invalidity of the transaction in more detail.

Table 14. Possible values of varying data type *UnknownTransaction*.

Id	Description	Type	Reject
0	Could not look up 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

C.6.2. `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 10.

C.6.3. `BlockBuilder_inherent_extrinisics`:

Generates the inherent extrinsics, which are explained in more detail in Section 2.3.3.. This function takes a SCALE-encoded hash table as defined in Definition 182 and returns an array of extrinsics. The Polkadot Host must submit each of those to the `BlockBuilder_apply_extrinsic`, described in Section C.6.1.. This procedure is outlined in Build-Block.

Arguments

- A Inherents-Data structure as defined in Definition 15.

Return

- A byte array of varying sizes containing extrinisics. Each extrinsic is a byte array of varying size.

C.6.4. `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 10 and the block body as described in Definition 13.
- A Inherents-Data structure as defined in Definition 15.

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 Definition 15 containing any errors created by this Runtime function.

C.7. Module TaggedTransactionQueue

> **NOTE**
> This section describes **Version 2** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.7.1. `TaggedTransactionQueue_validate_transaction`

This entry is invoked against extrinsics submitted through a transaction network message (Section 4.8.5.) or by an off-chain worker through the Host API (Section B.6.2.).

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 C.4.2..

Arguments

- The source of the transaction as defined in Definition 217.
- A byte array that contains the transaction.
- The hash of the parent of the block that the transaction is included in.

Definition 217. TransactionSource

TransactionSource is an enum describing the source of a transaction and can have one of the following values:

Table 15. 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 181 which contains the type *ValidTransaction* as defined in Definition 218 on success and the type *TransactionValidityError* as defined in Definition 214 on failure.

Definition 218. ValidTransaction

ValidTransaction is a tuple that contains information concerning a valid transaction.

Table 16. 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 64-bit integer
Requires	List of tags specifying extrinsics which should be applied before the current extrinsics 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

:::

INFO

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.

INFO

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.

C.8. Module OffchainWorkerApi

NOTE

This section describes **Version 2** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

Does not require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.8.1. `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 10.

Return

- None.

C.9. Module ParachainHost

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

C.9.1. `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.

C.9.2. `ParachainHost_validator_groups`

Returns the validator groups (Definition 126) used during the current session. The validators in the groups are referred to by the validator set Id (Definition 69).

Arguments

- None

Return

- An array of tuples, T, of the following format:

$$T = (I, G)$$

$$I = (v_n, \dots v_m)$$

$$G = (B_s, f, B_c)$$

where

- I is an array of the validator set Ids (Definition 69).
- 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.

C.9.3. `ParachainHost_availability_cores`

Returns information on all availability cores (Definition 125).

Arguments

- None

Return

- An array of core states, S, of the following format:

$$S = \begin{cases} 0 & \rightarrow & C_o \\ 1 & \rightarrow & C_s \\ 2 & \rightarrow & \phi \end{cases}$$

$$C_o = (n_u, B_o, B_t, n_t, b, G_i, C_h, C_d)$$

$$C_s = (P_id, C_i)$$

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 180) 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 180) 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 (Definition 131). 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 (Definition 126) 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 96).
- C_i is an *Option* value (Definition 180) which can contain the collators public key indicating who should author the block.

C.9.4. `ParachainHost_persisted_validation_data`

Returns the persisted validation data for the given parachain Id and a given occupied core assumption.

Arguments

- The parachain Id (Definition 124).
- An occupied core assumption (Definition 219).

Return

- An *Option* value (Definition 180) which can contain the persisted validation data (Definition 220). 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 219. Occupied Core Assumption

An 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:

$$A = \begin{cases} 0 & \rightarrow & \phi \\ 1 & \rightarrow & \phi \\ 2 & \rightarrow & \phi \end{cases}$$

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 220. 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 (Definition 123).
- 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 C.9.4.).

C.9.5. `ParachainHost_assumed_validation_data`

Returns the persisted validation data for the given parachain Id along with the corresponding Validation Code Hash. Instead of accepting validation about para, matches the validation data hash against an expected one and yields `None` if they are unequal.

Arguments

- The Parachain Id (Definition 124).
- Expected Persistent Validation Data Hash (Definition 220).

Return

- An *Option* value (Definition 180) which can contain the pair of persisted validation data (Definition 220) and Validation Code Hash. The value is `None` if the parachain Id is not registered or the validation data hash does not match the expected one.

C.9.6. `ParachainHost_check_validation_outputs`

Checks if the given validation outputs pass the acceptance criteria.

Arguments

- The parachain Id (Definition 124).
- The candidate commitments (Definition 97).

Return

- A boolean indicating whether the candidate commitments pass the acceptance criteria.

C.9.7. `ParachainHost_session_index_for_child`

Returns the session index that is expected at the child of a block.

> ⚠ **CAUTION**
> TODO clarify session index

Arguments

- None

Return

- A unsigned 32-bit integer representing the session index.

C.9.8. `ParachainHost_validation_code`

Fetches the validation code (Runtime) of a parachain by parachain Id.

Arguments

- The parachain Id (Definition 124).
- The occupied core assumption (Definition 219).

Return

- An *Option* value (Definition 180) containing the full validation code in a byte array. This value is empty if the parachain Id cannot be found or the assumption is wrong.

C.9.9. `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 180) containing the full validation code in a byte array. This value is empty if the parachain Id cannot be found or the assumption is wrong.

C.9.10. `ParachainHost_validation_code_hash`

Returns the validation code hash of a parachain.

Arguments

- The parachain Id (Definition 124).
- An occupied core assumption (Definition 219).

Return

- An *Option* value (Definition 180) containing the hash value of the validation code. This value is empty if the parachain Id cannot be found or the assumption is wrong.

C.9.11. `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 (Definition 124).

Return

- An *Option* value (Definition 180) containing the committed candidate receipt (Definition 94). This value is empty if the given parachain Id is not assigned to an occupied availability core.

C.9.12. `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:

$$E = \begin{cases} 0 & \rightarrow & d \\ 1 & \rightarrow & d \\ 2 & \rightarrow & (C_r, h, I_c) \end{cases}$$

$$d = (C_r, h, I_c, G_i)$$

where

- E specifies 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 94).
- h is the parachain head data (Definition 123).
- I_c is the index of the availability core as can be retrieved in Section C.9.3. 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 (Definition 126) that is responsible of backing the candidate.

C.9.13. `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 180) which can contain the session info structure, S, of the following format:

$$S = (A, D, K, G, c, z, s, d, x, a)$$

$$A = (v_n, \dots v_m)$$

$$D = (v_n, \dots v_m)$$

$$K = (v_n, \dots v_m)$$

$$G = (g_n, \dots g_m)$$

$$g = (A_n, \dots A_m)$$

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 C.9.1.).
- 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 C.9.1.). 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 C.9.1.).
- G indicates the validator groups in shuffled order.
- v_n is public key of the authority.
- A_n is the authority set Id (Definition 69).
- 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.

C.9.14. `ParachainHost_dmq_contents`

Returns all the pending inbound messages in the downward message queue for a given parachain.

Arguments

- The parachain Id (Definition 124).

Return

- An array of inbound downward messages (Definition 128).

C.9.15. `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 (Definition 124).

Return

- An array of inbound HRMP messages (Definition 130).

C.9.16. `ParachainHost_on_chain_votes`

Returns disputes relevant from on-chain, backing votes, and resolved disputes.

Arguments

- None

Return

- An *Option* (Definition 180) type which can contain the scraped on-chain votes data (Definition 221).

Definition 221. Scraped On Chain Vote

Contains the scraped runtime backing votes and resolved disputes.

The scraped on-chain votes data, $SOCV$, is a data structure of the following format:

$$SOCV = (S_i, BV, d)$$
$$BV = [C_r, [(i, a)]]$$

where:

- S_i is the u32 integer representing the session index in which the block was introduced.
- BV is the set of backing validators for each candidate, represented by its candidate receipt (Definition 94). Each candidate C_r has a list of (i, a), the pair of validator index and validation attestations (Definition 93).
- d is a set of dispute statements (Section 8.7.2.1.). Note that the above BV is unrelated to the backers of the dispute candidates.

> **⚠ CAUTION**
> PVF Pre-Checker subsystem is still Work-in-Progress, hence the below APIs are subject to change.

C.9.17. `ParachainHost_pvfs_require_precheck`

This runtime API fetches all PVFs that require pre-checking voting. The PVFs are identified by their code hashes. As soon as the PVF gains the required support, the runtime API will not return the PVF anymore.

Arguments

- None

Return

- A list of validation code hashes that require prechecking of votes by validators in the active set.

C.9.18. `ParachainHost_submit_pvf_check_statement`

This runtime API submits the judgment for a PVF, whether it is approved or not. The voting process uses unsigned transactions. The check is circulated through the network via gossip, similar to a normal transaction. At some point, the validator will include the statement in the block, where it will be processed by the runtime. If that was the last vote before gaining the super-majority, this PVF would not be returned by `pvfs_require_precheck` (Section C.9.17.) anymore.

Arguments

- A PVF pre checking statement (Definition 222) to be submitted into the transaction pool.
- Validator Signature (Definition 93).

Return

- None

Definition 222. PVF Check Statement

This is a statement by the validator who ran the pre-checking process for a PVF. A PVF is identified by the *ValidationCodeHash*. The statement is valid only during a single session, specified in the `session_index`.

The PVF Check Statement S_{pvf}, is a datastructure of the following format:

$$S_{pvf} = (b, VC_H, S_i, V_i)$$

where:

- b is a boolean denoting if the subject passed pre-checking.
- VC_H is the validation code hash.
- S_i is a u32 integer representing the session index.
- V_i is the validator index (Definition 93).

C.9.19. `ParachainHost_disputes`

This runtime API fetches all on-chain disputes.

Arguments

- None

Return

- A list of (SessionIndex, CandidateHash, DisputeState).

> **CAUTION**
> TODO clarify DisputeState

C.9.20. `ParachainHost_executor_params`

This runtime API returns execution parameters for the session.

Arguments

- Session Index

> **CAUTION**
> TODO clarify session index

Return

- Option type of Executor Parameters.

> **CAUTION**
> TODO clarify Executor Parameters

C.10. Module GrandpaApi

> **NOTE**
> This section describes **Version 2** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.10.1. `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 33. Any future authority changes get tracked via Runtime-to-consensus engine messages, as described in Section 3.3.2..

Arguments

- None.

Return

- An authority list as defined in Definition 33.

C.10.2. `GrandpaApi_current_set_id`

This entry fetches the list of GRANDPA authority set IDs (Definition 69). Any future authority changes get tracked via Runtime-to-consensus engine messages, as described in Section 3.3.2..

Arguments

- None.

Return

- An authority set ID as defined in Definition 69.

C.10.3. `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 76. 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:

$$G_{\text{Ep}} = (\text{id}_{\mathbb{V}}, e, r, A_{\text{id}}, B_h^1, B_n^1, A_{\text{sig}}^1, B_h^2, B_n^2, A_{\text{sig}}^2)$$

$$e = \begin{cases} 0 & \text{Equivocation at prevote stage} \\ 1 & \text{Equivocation at precommit stage} \end{cases}$$

  where

  - $mathrm{id}_{\mathbb{V}}$ is the authority set id as defined in Definition 69.

- e indicates the stage at which the equivocation occurred.
- r is the round number the equivocation occurred.
- $A_{mathrm{id}}$ 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.
- $A_{\{mathrm\{sig\}\}}^1$ 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.
- $A_{\{mathrm\{sig\}\}}^2$ is the equivocators signature of the second vote.
- A proof of the key owner in an opaque form as described in Section C.10.4..

Return

- A SCALE encoded *Option* as defined in Definition 180 containing an empty value on success.

C.10.4. `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 C.10.3..

Arguments

- The authority set id as defined in Definition 69.
- The 256-bit public key of the authority.

Return

- A SCALE encoded *Option* as defined in Definition 180 containing the proof in an opaque form.

C.11. Module BabeApi

> **NOTE**
> This section describes **Version 2** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialized_block` (Section C.4.3.) to be called beforehand.

C.11.1. `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 17. 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 55.	Tuple containing two unsigned 64bit integers
GenesisAuthorities	The authority list for the genesis epoch as defined in Definition 33.	Array of tuples containing a 256-bit byte array and an unsigned 64bit integer
Randomness	The randomness for the genesis epoch	32-byte array
SecondarySlot	Whether this chain should run with a round-robin-style secondary slot and if this secondary slot requires the inclusion of an auxiliary VRF output (Section 5.2.).	A one-byte enum as defined in Definition 54 as 2_{nd}.

C.11.2. `BabeApi_current_epoch_start`

Finds the start slot of the current epoch.

Arguments

- None.

Return

- A unsigned 64-bit integer indicating the slot number.

C.11.3. `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 an unsigned 64-bit integer representing the starting slot of the epoch.
- d is an unsigned 64-bit integer representing the duration of the epoch.
- A is an authority list as defined in Definition 33.
- r is a 256-bit array containing the randomness for the epoch as defined in Definition 67.

C.11.4. `BabeApi_next_epoch`

Produces information about the next epoch.

Arguments

- None.

Return

- Returns the same data structure as described in Section C.11.3..

C.11.5. `BabeApi_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 C.11.6..

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 180 containing the proof in an opaque form.

C.11.6. `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.

> **INFO**
> If there are more than two blocks that 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:

$$B_{mathrm{Ep}} = \left(A_{mathrm{id}}, s, h_1, h_2\right)$$

  where

  - $A_{mathrm{id}}$ is the public key of the equivocator.
  - s is the slot as described in Definition 50 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 C.11.5..

Return

- A SCALE encoded *Option* as defined in Definition 180 containing an empty value on success.

C.12. Module AuthorityDiscoveryApi

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require (Section Section C.4.3.) to be called beforehand.

C.12.1. `AuthorityDiscoveryApi_authorities`

A function that helps to discover authorities.

Arguments

- None.

Return

- A byte array of varying size containing 256-bit public keys of the authorities.

C.13. Module SessionKeys

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.13.1. `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 180 containing an array of varying sizes indicating the seed.

Return

- A byte array of varying size containing the encoded session keys.

C.13.2. `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:

$$\left(k, k_{mathrm\{id\}}\right)$$

where k is an array of varying sizes containing the raw public key and $k_{mathrm\{id\}}$ is a 4-byte array indicating the key type.

C.14. Module AccountNonceApi

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.14.1. `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 C.1.1..

Arguments

- The 256-bit public key of the account.

Return

- A 32-bit unsigned integer indicating the nonce of the account.

C.15. Module TransactionPaymentApi

> **NOTE**
> This section describes **Version 2** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility.

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

C.15.1. `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 C.1.1..

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 a class is a varying data (Definition 178) 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.

C.15.2. `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 C.1.1..

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 180 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.

C.16. Module TransactionPaymentCallApi

All calls in this module require `Core_initialize_block` (Section C.4.3.) to be called beforehand.

> **CAUTION**
> TODO clarify differences between *RuntimeCall* and *Extrinsics*

C.16.1. `TransactionPaymentCallApi_query_call_info`

Query information of a dispatch class, weight, and fee of a given encoded `Call`.

Arguments

- A byte array of varying sizes containing the `Call`.
- The length of the Call.

Return

- A data structure of the following format:

$$(w, c, f)$$

where:

- w is the weight of the call.
- c is the "class" of the call, where a class is a varying data (Definition 178) type defined as:

$$c = \begin{cases} 0 & \text{Normal dispatch} \\ 1 & \text{Operational dispatch} \\ 2 & \text{Mandatory dispatch, which is always included regardless of their weight} \end{cases}$$

- f is the partial-fee of the call. This does not include a tip or anything else that depends on the signature.

C.16.2. `TransactionPaymentCallApi_query_call_fee_details`

Query the fee details of a given encoded `Call` including tip.

Arguments

- A byte array of varying sizes containing the `Call`.
- The length of the `Call`.

Return

- A data structure of the following format:

$$(f, t)$$

where:

- f is a SCALE encoded as defined in Definition 180 containing the following data structure:

$$f = (f_b, f_l, f_a)$$

where:

- f_b is the minimum required fee for the `Call`.
- f_l is the length fee, the amount paid for the encoded length (in bytes) of the `Call`.
- 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.

C.17. Module Nomination Pools

> **NOTE**
> This section describes **Version 1** of this API. Please check `Core_version` (Section C.4.1.) to ensure compatibility. Currently supports only one RPC endpoint.

C.17.1. `NominationPoolsApi_pending_rewards`

Runtime API for accessing information about the nomination pools. Returns the pending rewards for the member that the Account ID was given for.

Arguments

- The account ID as a SCALE encoded 32-byte address of the sender (Definition 134).

Return

- The SCALE encoded balance of type `u128` representing the pending reward of the account ID. The default value is Zero in case of errors in fetching the rewards.

C.17.2. `NominationPoolsApi_points_to_balance`

Runtime API to convert the number of points to balances given the current pool state, which is often used for unbonding.

Arguments

- An unsigned 32-bit integer representing Pool Identifier
- An unsigned 32-bit integer Points

Return

- An unsigned 32-bit integer Balance

C.17.3. `NominationPoolsApi_balance_to_points`

Runtime API to convert the given amount of balances to points for the current pool state, which is often used for bonding and issuing new funds in to the pool.

Arguments

- An unsigned 32-bit integer representing Pool Identifier
- An unsigned 32-bit integer Balance

Return

- An unsigned 32-bit integer Points

Glossary

P_n

A path graph or a path of n nodes.

$(b_0, b_1, \ldots, b_{n-1})$

A sequence of bytes or byte array of length n

$\mathbb{B}_n$

A set of all byte arrays of length n

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

A non-negative integer in base 256

$B = (b_0, b_1, \ldots, b_n)$

The little-endian representation of a non-negative interger $I = (B_n \ldots B_0)_{256}$ such that $b_i := B_i$

Enc_{LE}

The little-endian encoding function.

C

A blockchain is 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

$\mathrm{CHAIN}(B)$

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

$\mathrm{SubChain}(B', B)$

The subgraph of $Chain(B)$ path graph containing both B and B'.

$\mathbb{C}_B(BT)$

The set of all subchains of BT rooted at block B.

$\mathbb{C}, \mathbb{C}(BT)$

$\mathbb{C}_G(BT)$ i.e. the set of all chains of BT rooted at genesis block

$\text{Longest-Chain}(BT)$

The longest sub path graph of BT i.e. $C : |C| = \max_{C_i \in \mathbb{C}} |C_i|$

$\text{Longest-Path}(BT)$

The longest sub path graph of $(P)BT$ with earliest block arrival time

$\text{Deepest-Leaf}(BT)$

$\text{HeadLongest-Path}(BT)$ i.e. the head of $\text{Longest-Path}(BT)$

$B > B'$

B is a descendant of B' in the block tree

$\text{StoredValue}(k)$

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 entries.

$\text{KeyEncode}(k)$

The function to encode keys for labeling branches of the trie.

$\mathscr{N}$

The set of all nodes in the Polkadot state trie.

N

An individual node in the trie.

$\mathscr{N}_b$

A branch node of the trie which has at least one and at most 16 children

$\mathscr{N}_l$

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} representing 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 indicates 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 14.

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 integer 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 auxiliary field in the 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

$Body(B)$

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 votes in the finality protocol

k_v^{pr}

The private key of voter v

v_{id}

The public key of voter v

$\mathbb{V}_B, \mathbb{V}$

The set of all GRANDPA voters for at block B

GS

GRANDPA protocol state consisting of the set of voters, the 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 vote to block B, broadcasted during the specified stage of finality round r

$\mathscr{E}^{r,stage}$

The set of all equivocator voters in sub-round ‘‘stage" of round r

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

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

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

A tuple of values A_i's each of different type

Varying Data Types $\mathscr{T} = \{T_1, \ldots, T_n\}$

A data type representing any of varying types $T_1, \ldots, T_n$.

$S := A_1, \ldots, A_n$

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