Polkadot Runtime Environment
Protocol Specification

January 14, 2018

1 Conventions and Definitions

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

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

Definition 3 *radix r tree is a variant of a trie in which:*

- *Every node has at most r children where $r = 2^x$ for some x;*

- *Each node that is the only child of a parent, which does not represent a valid key is merged with its parent.*

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

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

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

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

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

Notation 1 *We represent the concatenation of byte arrays $a := (a_0, \ldots, a_n)$ and $b := (b_0, \ldots, b_m)$ by:*

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

Definition 5 *For a given byte b the **bitwise representation** of b is defined as*

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

where

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

2 Block

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

2.1 Block Header

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

Definition 6 *The **header of block B**,* Head(B) *is a 5-tuple containing the following elements:*

- **parent_hash:** *is the 32-byte Blake2s hash of the header of the parent of the block indicated henceforth by H_p.*

- **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 block has number 0.*

- **state_root:** *formally indicated as H_r is the root of the Merkle trie, whose leaves implement the storage for the system.*

- **extrinsics_root:** *is the root of the Merkle trie, whose leaves represent individual extrinsics being validated in this block. 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. Polkadot RE does not impose any limitation or specification for this field. Essentially, it can be a byte array of any length. This field is indicated as H_d*

Definition 7 *The **Block Header Hash of Block** B, $H_h(b)$, is the hash of the header of block B encoded by simple codec:*

$$H_b(b) := \mathrm{Blake}\,2s(\mathrm{Enc}_{\mathrm{SC}}(\mathrm{Head}(B)))$$

2.2 Justified Block Header

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

- **block_header** the complete block header as defined in Section 2.1 and denoted by $\text{Head}(B)$.

- **justification**: as defined by the consensus specification indicated by $\text{Just}(B)$ link this to its definition from consensus.

- **authority Ids**: This is the list of the Ids of authorities, which have voted for the block to be stored and is formally referred to as $A(B)$. An authority Id is 32bit.

2.3 Extrinsics

Each block also contains a list of extrinsics. Polkadot RE does not specify or limit the internal of each extrinsics beside the fact that each extrinsics is a blob of encoded data. The extrinsics_root should correspond to the root of the Merkle trie, whose leaves are made of the block extrinsics list.

3 Runtime

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

$$b = 3a, 63, 6f, 64, 65$$

which is the byte array of ASCII representation of string ":code" (see Section 10). For any call to the runtime, Polkadot RE makes sure that it has most updated runtime as calls to runtime have the ability to change the runtime code.

The initial runtime code of the chain is embedded as an extrinsics into the chain initialization JSON file and is submitted to Polkadot RE (see Section 9).

Subsequent calls to the runtime have the ability to call the storage API (see Section 12) to insert a new Wasm blob into runtime storage slot to upgrade the runtime.

3.1 Entries into Runtime

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

```
(export "version" (func $version))
(export "authorities" (func $authorities))
(export "execute_block" (func $execute_block))
```

Table 1: Snippet to export entries
into tho Wasm runtime module

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

3.1.1 version

This entry receives no argument, it returns the vesion data encoded in ABI format described in Section 3.3 containing the following data:

Name	Type	Description
spec_name	String	runtime identifier
impl_name	String	the name of the implementation (e.g. C++)
authoring_version	32-bit non-negative integer	the version of the authorship interface
spec_version	32-bit non-negative integer	the version of the runtime specification
impl_version	32-bit non-negative integer	the version of the runtime implementation
apis	ApisVec	List of supported API

Table 2: Detail of the version data
type returns from runtime `version`
function

3.1.2 authorities

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

3.1.3 execute_block

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

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

Table 3: Detail of the data exe-
cute_block returns after execution

3.2 Code Executor

Polkadot RE provide a Wasm Virtual Machine (VM) to run the runtime. The Wasm VM exposes the Polkadot RE API to the Runtime. And execute the Runtime as a Wasm module.

3.3 ABI Encoding between Runtime and the Runtime Enviornment

All data exchanged between Polkadot RE and the runtime is encoded using SCALE codec described in Section 8.1.

4 Network API

4.1 Block Submission and Validation

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

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

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

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

IMPORT-AND-VALIDATE-BLOCK$(B, \text{Just}(B))$

1   VERIFY-BLOCK-JUSTIFICATION$(B, \text{Just}(B))$
2   Verify $H_{p(B)} \in$ Blockchain.
3   State-Changes = Runtime.(B)
4   UPDATE-WORLD-STATE(State-Changes)

5 State Storage and the Storage Trie

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

5.1 Accessing System Storage

Polkadot RE implements various functions to facilitate access to system storage for the runtime. Section 12 lists all of those functions. Here we define the most essential ones which are also used by the Polkadot RE.

Definition 8 *StateRead and StateWrite function provide basic access to the State Storage:*

$$v = \text{StateRead}(k)$$

$$\text{StateWrite}(k, v)$$

where v and k are byte arrays.

To authenticate the state of the system, the stored data needs to be re-arranged and hashed in a *radix 16 tree* also known as *base-16 modified Merkle Patricia Tree*, which hereafter we will refer to as the **Trie**, in order to compute the hash of the whole state storage consistently and efficiently at any given time.

As well, modification has been made in the storing of the the nodes' hash in the Merkle Tree structure to save space on entries storing small entries.

Because the Tri is used to compute the *state root*, H_r, (see Definition 6), which is used to authenticate the validity of the state database, Polkadot RE follows a rigorous encoding algorithm to compute the values stored in the trie nodes to ensure that the computed Merkle hash, H_r, matches across clients.

5.2 The General Tree Structure

As the trie is a radix 16 tree, in this sense, 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.

To identify the node corresponding to a key value, k, first we need to encode k in a uniform way:

Definition 9 *The for the purpose labeling the branches of the Trie key k is encoded to k_{enc} using KeyEncode functions:*

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

such that:

$$\text{KeyEncode}(k) : \begin{cases} \mathbb{B} & \to & \text{Nibbles}_4 \\ k := (b_1, \ldots, b_n) := & \mapsto & (b_1^1, b_1^2, b_2^1, b_2^2, \ldots, b_n^1, b_n^2) \\ & & := (k_{\text{enc}_1}, \ldots, k_{\text{enc}_{2n}}) \end{cases}$$

where Nibble_4 is the set of all nibbles of 4-bit arrays and b_i^1 and b_i^2 are 4-bit nibbles, which are the little endian representations of b_i:

$$(b_i^1, b_i^2) := (b_1 \bmod 16, b_2/16)$$

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

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

5.3   The Merkle proof

To prove the consistency of the state storage across the network and its modifications efficiently, the Merkle hash of the storage trie needs to be computed rigorously.

The Merkle hash of the trie is computed recursively. As such, hash value of each node depends on the hash value of all its children and also on its value. Therefore, it suffices to define how to compute the hash value of a typical node as a function of the hash value of its children and its own value.

Definition 10 *Suppose node N of storage state trie has key value k_N, and parent key value of $k_{P(N)}$, such that:*

$$\mathrm{KeyEncode}(k_N) = (k_{\mathrm{enc}_1}, \ldots, k_{\mathrm{enc}_{i-1}}, k_{\mathrm{enc}_i}, \ldots, k_{\mathrm{enc}_{2n}})$$

and

$$\mathrm{KeyEncode}(k_{P(N)}) = (k_{\mathrm{enc}_1}, \ldots, k_{\mathrm{enc}_{i-1}})$$

We define

$$\mathrm{pk}_N := (k_{\mathrm{enc}_i}, \ldots, k_{\mathrm{enc}_{2n}})$$

*to be the **the partial key** of N.*

Definition 11 *For a trie node N, **Node Prefix** function is a value specifying the node type as follows:*

$$\mathrm{NodePrefix}(N) := \begin{cases} 1 & N \text{ is a leaf node} \\ 254 & N \text{ is a branch node without value} \\ 255 & N \text{ is a branch node with value} \end{cases}$$

Definition 12 *For a given node N, with partial key of pk_N and Value v, the **encoded representation** of N, formally referred to as $\mathrm{Enc}_{\mathrm{Node}}(N)$ is determined as follows, in case which:*

- *N is a leaf node:*

$$\mathrm{Enc}_{\mathrm{Node}}(N) := \mathrm{Enc}_{\mathrm{len}}(N) \,||\, \mathrm{HPE}(\mathrm{pk}_N) \,||\, \mathrm{Enc}_{\mathrm{SC}}(v)$$

- *N is a branch node:*

$$Enc_{Node}(N) := \\ \mathrm{NodePrefix}(N) \,||\, \mathrm{ChildrenBitmap}(N) \,||\, \mathrm{HPE}_{\mathrm{PC}}(v) \,||\, \mathrm{Enc}_{\mathrm{SC}}(\mathrm{Enc}_{\mathrm{Node}}) \,|| \\ \mathrm{Enc}_{\mathrm{SC}}(N_{C_1}) \ldots \mathrm{Enc}_{\mathrm{SC}}(N_{C_n})$$

Where $N_{C_1} \ldots N_{C_n}$ with $n \leqslant 16$ are the children nodes of N.

Definition 13 *For a given node N, the **Merkle value** of N, denoted by $H(N)$ is defined as follows:*

$$H : \mathbb{B} \to \bigcup_{i=0}^{32} \mathbb{B}_i$$
$$H(N) : \begin{cases} \mathrm{Enc}_{\mathrm{Node}}(N) & \| \mathrm{Enc}_{\mathrm{Node}}(N) \| < 32 \\ \mathrm{Hash}(\mathrm{Enc}_{\mathrm{Node}}(N)) & \| \mathrm{Enc}_{\mathrm{Node}}(N) \| \geqslant 32 \end{cases}$$

6 Extrinsics trie

To validate that Extrinsics data are stored in a block across clients, Polkadot RE uses the same trie structure as for the state storage described in Section 5.3 to generate the Merkle proof.

7 Consensus Engine

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

7.1 Block Tree

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

Definition 14 *The **Block Tree** of a blockchain is the union of all different versions of the blockchain observered by all the node in the system such as every such block is a node in the graph and B_1 is connected to B_2 if B_1 is parent of B_2.*

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

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

Definition 15 *We say **B is descendant of** B', formally noted as $B > B'$ if B is a descendent of B' in the block tree.*

7.2 Block Production

7.3 Finality

Polkadot RE uses GRANDPA Finality protocol [**?**] to finalize blocks. Finality is obtained by consequtive rounds of voting by validator nodes. Validators executes GRANDPA finality process in parallel to Block Production as an indepenedent service. In this section we describe the different functions that GRANDPA service is supposed to perform in order to successfully participates in block finalization process.

7.3.1 Priliminaries

Definition 16 *A **Grandpa Voter**, v, is represented by a key pair (k_v^{pr}, k_v) where k_v^{pr} represents its private key, is a node running Grandpa protocol and broadcast votes to finilse blocks in a Polkadot RE based chain. The **set of all Grandpa voters** is indicated by $\mathbb{V}$.*

Definition 17 *__Grandpa state__, GS, is defined as*

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

where:

    $\mathbb{V}$*: is the set of voters.*
    $\mathbb{V}_{\mathrm{id}}$*: is an incremental counter tracking membership changes in V.*
    r: *is voting round number.*

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

Definition 18 *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 block hash and block number defined in Definitions 6 and 7 respectively.

Definition 19 *Voters engage in maximum two sub-rounds of voting for each round r. The first sub-round is called **pre-vote** and the second sub-round is called **pre-commit**.*
    By $V_v^{r,\mathrm{pv}}$ and $V_v^{r,\mathrm{pc}}$ we refere to the vote casted by voter v in round r (for block B) during the pre-vote and the pre-commit sub-round respectively.

    The GRANDPA protocol dictates how an honset voter should vote in each sub-round which is described in Algorithm 7.3.3.

Definition 20 *A vote casted by voter v should be broadcasted as a **message** $M_v^{r,\mathrm{stage}}$ to the network by voter v with the following structure:*

$$M_v^{r,\mathrm{stage}} := \mathrm{Enc}_{\mathrm{SC}}(r, \mathrm{id}_{\mathbb{V}}, \mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, \mathrm{Sig}_{\mathrm{ED\,25519}}(\mathrm{Enc}_{\mathrm{SC}}(\mathrm{stage}, V_v^{r,\mathrm{stage}}, r, V_{\mathrm{id}}), v_{\mathrm{id}})$$

Where:

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

Definition 21 *A vote $V_v^{r,\text{stage}} = V(B)$ is **invalid** if*

- *$H(B)$ does not correspond to a valid block.*
- *B is not a (eventual) descendent of a previously finalized block.*
- *$M_v^{r,\text{stage}}$ does not bear a vaid signature.*
- *$\text{id}_{\mathbb{V}}$ does not match the current $\mathbb{V}$.*
- *If voter v has broadcasted two otherwise valid votes during one sub-round.*

Definition 22 *For validator v, **the set of observed direct vote for Block B in round** r, formally denoted by $\text{VD}_{\text{obs}(v)}^{r,\text{stage}}(B)$ is equal to the set of valid votes $V_{v_i}^{r,\text{stage}}$ casted in round r and received by v such that $V_{v_i}^{r,\text{stage}} = V(B)$.*

Definition 23 *We refer to **the set of total vote observed by voter v in sub-round** stage **of round** r by $V_{\text{obs}(v)}^{r,\text{stage}}$.*

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

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

*The **total observed vote for Block B in round** r is defined to be the size of that set, formally:*

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

Definition 24 *The current **pre-voted** block $B_v^{r,\text{pv}}$ is the block with*

$$H_n(B_v^{r,\text{pv}}) = \text{Max}(|H_n(B)| \text{ for all } B : |V_{\text{obs}(v)}^{r,\text{pv}}(B)| \geqslant 2/3|\mathbb{V}|)$$

*Similarly the **pre-commited** block $B_v^{r,\text{pc}}$ is the block with*

$$H_n(B_v^{r,\text{pc}}) = \text{Max}(|H_n(B)| \text{ for all } B : |V_{\text{obs}(v)}^{r,\text{pc}}(B)| \geqslant 2/3|\mathbb{V}|).$$

*Let B' be any block such that $B' \leqslant B_v^{r,\text{stage}}$. We define the $C_v^{r,\text{stage}}$, **the chosen block of** $V_{\text{obs}(v)}^{r,\text{stage}}$ as follows:*

$$C_v^{r,\text{stage}} := \begin{cases} B & \text{such that } H_n(B) = \text{Max}\{H_n(B')\} & \text{if } |V_{\text{obs}(v)}^{r,\text{pv}}(B_v^{r,\text{pv}})| > 2/3|\mathbb{V}| \\ \phi & & \text{otherwise} \end{cases}$$

Definition 25 *We say round r is **unfinalizable,** If for all $B' \geqslant B_v^{r,\text{pv}}$:*

$$|V_{\text{obs}(v)}^{r,\text{pc}}| - |V_{\text{obs}(v)}^{r,\text{pc}}(B')| \geqslant \frac{1}{3}|\mathbb{V}|$$

7.3.2 Initiating the GRANDPA state

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

As instructed in Algortihm 7.3.2, whenever the membership of Grandpa voters changes, r is set to 0 and V_{id} needs to be incremented.

JOIN-LEAVE-GRANDPA-VOTERS (V)
1   $r \leftarrow 0$
2   $V_{\text{id}} \leftarrow$ ReadState($'\text{AUTHORITY\_SET\_KEY}'$)
3   $V_{\text{id}} \leftarrow V_{\text{id}} + 1$
2   EXECUTE-ONE-GRANDPA-ROUND(r)
Each voter should run Algorithm 7.3.3 to verify that a round is completable

7.3.3 Voting process in round r

At each round r, a randomly selected

For each round r, an honest voter v must participate in voting process by following Algorithm 7.3.3.

PLAY-GRANDPA-ROUND(r)
1   $t_{r,v} \leftarrow$ Time
2   primary $\leftarrow$ DERIVE-PRIMARY
4   **if** $v =$ primary:
5   BROADCAST($M_v^{r-1,\text{fin}}()$)
6   **else**
9   RECEIVE-MESSAGES(**until** Time $\geqslant t_{r,v} + 2 \times T$ **or** COMPLETABLE(r))
10      $L \leftarrow$ RECEIVED-AS-FINAL() **or** BEST-FINAL-CANDIDATE(r-1):
11   **if** RECEIVED($M_{v_{\text{primary}}}^{r,\text{pv}}(B)$) **and** $B_v^{r,\text{pv}} \geqslant B > L$:
12          $N \leftarrow B$
13   **else**
14          $N \leftarrow B' : H_n(B') = \max\{H_n(B') : B' > L\}$
15 BROADCAST($M_v^{r,\text{pv}}(N)$)
16 RECEIVE-MESSAGES(**until** $B_v^{r,\text{pv}} \geqslant L$ **and** (Time $\geqslant t_{r,v} + 4 \times T$ **or** COMPLETABLE(r)))
12 BROADCAST($M_v^{r,\text{pc}}$(BEST-FINAL-CANDIDATE(r))
13 PLAY-GRANDPA-ROUND($r + 1$)
BEST-FINAL-CANDIDATE(r)
1   $\mathcal{C} \leftarrow \{B'|B' \leqslant B_v^{r,\text{vc}} : |V_v^{r,\text{pc}}| - |V_v^{r,\text{pc}}(B')| \leqslant 1/3|\mathbb{V}|\}$
2   **if** $\mathcal{C} = \phi$:
3   **return** ϕ
4   **else**
5   **return** $E \in \mathcal{C} : H_n(E) := \max\{H_n(B') : B' \in \mathcal{C}\}$
COMPLETABLE(r)

1   **if** $\exists B' \leqslant B_v^{r,\mathrm{pv}}$ and $|V_v^{r,\mathrm{pc}}| - |V_v^{r,\mathrm{pc}}(B)| \leqslant 1/3|\mathbb{V}|$
2   **return** TRUE
3   **elif** r is unfinalizable:
4   **return** TRUE
5   **else**
6   **return** FALSE
FINALIZEROUND(r)
1   $L \leftarrow$ LAST-FINALIZED-BLOCK
2   **if** $C^{r,\mathrm{pc}} \geqslant L$ and $V_{\mathrm{obs}(v)}^{r-1,\mathrm{pc}}(L) > 2/3|\mathcal{V}|$ and $C^{r,\mathrm{pv}}(P) \neq \phi$
3   LAST-FINALIZED-BLOCK$\leftarrow C^{r,\mathrm{pc}}$
4   **if** $M_v^{r,\mathrm{Fin}}(C^{r,\mathrm{pc}}) \notin$ RECEIVED-MESSAGES:
5   BROADCAST($M_v^{r,\mathrm{Fin}}(C^{r,\mathrm{pc}})$)

8   Auxilary Encodings

8.1   SCALE Codec

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

Definition 26 *The **SCALE codec** for **Byte array** A such that*

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

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

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

in which:

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

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

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

where:

$$m = l_1^7 \ldots l_1^3 l_1^2 + 4$$

Definition 27 *The **SCALE codec** for **Tuple** T such that:*

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

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

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

8.2 Hex Encoding

Practically it is more convenient and efficient to store and process data which is stored in a byte array. On the other hand, radix-16 tree keys are broken in 4-bits nibbles. Accordingly, we need a method to encode sequences of 4-bits nibbles into byte arrays canonically:

Definition 28 *Suppose that $\mathrm{PK} = (k_1, \ldots, k_n)$ is a sequence of nibbles, then*
$\mathrm{Enc}_{\mathrm{HE}}(\mathrm{PK}) :=$

$$\begin{cases} \mathrm{Nibbles}_4 & \to & \mathbb{B} \\ \mathrm{PK} = (k_1, \ldots, k_n) & \mapsto & \begin{cases} (0, k_1 + 16k_2, \ldots, k_{2i-1} + 16k_{2i}) & n = 2i \\ (k_1, k_2 + 16k_3, \ldots, k_{2i} + 16k_{2i+1}) & n = 2i + 1 \end{cases} \end{cases}$$

8.3 Partial Key Encoding

Definition 29 *Let N be a node in the storage state trie with Partial Key PK_N. We define the **Partial key length encoding** function, formally referred to as $\mathrm{Enc}_{\mathrm{len}}(N)$ as follows:*

$$\begin{array}{lll} \mathrm{Enc}_{\mathrm{len}}(N) & & := \\ \mathrm{NodePrefix}(N) & & + \\ \begin{cases} (\|\,\mathrm{PK}_N\,\|) & \mathrm{NisleafNode} \quad \& \quad \|\,\mathrm{PK}_N\,\| < 127 \\ (127)\|(\mathrm{LE}(\|\,\mathrm{PK}_N\,\| - 127)) & \mathrm{NisaleafNode} \quad \& \quad \|\,\mathrm{PK}_N\,\| \geqslant 127 \end{cases} \end{array}$$

where $\mathrm{NodePrefix}$ function is defined in Definition 11.

9 Genisis Block Specification

10 Predefined Storage keys

11 Runtime upgrade

12 Runtime API

Runtime API is a set of functions, which Polkadot RE exposes to Runtime in order to access Storage content and other external functions. Some of the functions are exposed to the runtime for efficiency reasons. Here is the list of the functions which Polkadot RE exposes to the runtime:

- ext_blake2_256

- ext_blake2_256_enumerated_trie_root

- ext_chain_id

- ext_child_storage_root

- ext_clear_child_storage

- ext_clear_prefix

- ext_clear_storage

- ext_ed25519_verify

- ext_exists_child_storage

- ext_free

- ext_get_allocated_child_storage

- ext_get_allocated_storage

- ext_get_child_storage_into

- ext_get_storage_into

- ext_kill_child_storage

- ext_malloc

- ext_print_hex

- ext_print_num

- ext_print_utf8

- ext_sandbox_instance_teardown

- ext_sandbox_instantiate

- ext_sandbox_invoke

- ext_sandbox_memory_get

- ext_sandbox_memory_new

- ext_sandbox_memory_set

- ext_sandbox_memory_teardown

- ext_set_child_storage

- ext_set_storage

- ext_storage_changes_root

- ext_storage_root

- ext_twox_128

- ext_twox_256

- ext_exists_storage