Telegram Open Network Blockchain

Nikolai Durov

July 21, 2024

Abstract

The aim of this text is to provide a detailed description of the Telegram Open Network (TON) Blockchain.

Introduction

This document provides a detailed description of the TON Blockchain, including its precise block format, validity conditions, TON Virtual Machine (TVM) invocation details, smart-contract creation process, and cryptographic signatures. In this respect it is a continuation of the TON whitepaper (cf. [3]), so we freely use the terminology introduced in that document.

Chapter **1** provides a general overview of the TON Blockchain and its design principles, with particular attention to the introduction of compatibility and validity conditions and the implementation of message delivery guarantees. More detailed information, such as the TL-B schemes that describe the serialization of all required data structures into trees or collections ("bags") of cells, is provided in subsequent chapters, culminating in a complete description of the TON Blockchain (shardchain and masterchain) block layout in Chapter **5**.

A detailed description of the elliptic curve cryptography used for signing blocks and messages, also accessible through TVM primitives, is provided in Appendix **A**. TVM itself is described in a separate document (cf. [4]).

Some subjects have intentionally been left out of this document. One is the Byzantine Fault Tolerant (BFT) protocol used by the validators to determine the next block of the masterchain or a shardchain; that subject is left for a forthcoming document dedicated to the TON Network. And although

this document describes the precise format of TON Blockchain blocks, and discusses the blockchain's validity conditions and serialized invalidity proofs,[1] it provides no details about the network protocols used to propagate these blocks, block candidates, collated blocks, and invalidity proofs.

Similarly, this document does not provide the complete source code of the masterchain smart contracts used to elect the validators, change the configurable parameters or get their current values, or punish the validators for their misbehavior, even though these smart contracts form an important part of the total blockchain state and of the masterchain block zero. Instead, this document describes the location of these smart contracts and their formal interfaces.[2] The source code of these smart contracts will be provided separately as downloadable files with comments.

Please note that the current version of this document describes a preliminary test version of the TON Blockchain; some minor details are likely to change prior to launch during the development, testing, and deployment phases.

[1] As of August 2018, this document does not include a detailed description of serialized invalidity proofs, because they are likely to change significantly during the development of the validator software. Only the general design principles for consistency conditions and serialized invalidity proofs are discussed.

[2] This is not included in the present version of this document, but will be provided in a separate appendix to a future revision.

Contents

1 Overview

This chapter provides an overview of the main features and design principles of the TON Blockchain. More detail on each topic is provided in subsequent chapters.

1.1 Everything is a bag of cells

All data in the blocks and state of the TON Blockchain is represented as a collection of *cells* (cf. [3, 2.5]). Therefore, this chapter begins with a general discussion of cells.

1.1.1. TVM cells. Recall that the TON Blockchain, as well as the TON Virtual Machine (TVM; cf. [4]), represents all permanently stored data as a *collection* or *bag* of so-called *cells*. Each cell consists of up to 1023 data bits and up to four references to other cells. Cyclic cell references are not allowed, so the cells are usually organized into *trees of cells*, or rather *directed acyclic graphs (DAGs) of cells*.[3] Any value of an abstract algebraic (dependent) data type may be represented (serialized) as a tree of cells. The precise way of representing values of an abstract data type as a tree of cells is expressed by means of a *TL-B scheme*.[4] A more thorough discussion of different kinds of cells may be found in [4, 3.1].

1.1.2. Application to TON Blockchain blocks and state. The above is particularly applicable to the blocks and state of the TON Blockchain, which also are values of certain (quite convoluted) dependent algebraic data types. Therefore, they are serialized according to various TL-B schemes (which are gradually presented throughout this document), and are represented as a collection or bag of cells.

1.1.3. The layout of a single cell. Each single cell consists of up to 1023 data bits and up to four references to other cells. When a cell is kept in memory, its exact representation is implementation-dependent. However,

[3] Completely identical cells are often identified in memory and in disk storage; this is the reason why trees of cells are transparently transformed into DAGs of cells. From this perspective, a DAG is just a storage optimization of the underlying tree of cells, irrelevant for most considerations.

[4] Cf. [4, 3.3.3–4], where an example is given and explained, pending a more complete reference

there is a standard representation of cells, useful, for instance, for serializing cells for file storage or network transmission. This "standard representation" or "standard layout" CellRepr(c) of a cell c consists of the following:

- Two *descriptor bytes* come first, sometimes denoted by d_1 and d_2. The first of these bytes d_1 equals (in the simplest case) the number of references $0 \leq r \leq 4$ in the cell. The second descriptor byte d_2 encodes the bit length l of the data part of the cell as follows: the first seven bits of d_2 equal $\lfloor l/8 \rfloor$, the number of complete data bytes present in the cell, while the last bit of d_2 is the *completion tag*, equal to one if l is not divisible by eight. Therefore,

$$d_2 = 2\lfloor l/8 \rfloor + [l \bmod 8 \neq 0] = \lfloor l/8 \rfloor + \lceil l/8 \rceil \tag{1}$$

  where $[A]$ equals one when condition A is true, and zero otherwise.

- Next, $\lceil l/8 \rceil$ data bytes follow. This means that the l data bits of the cell are split into groups of eight, and each group is interpreted as a big-endian 8-bit integer and stored into a byte. If l is not divisible by eight, a single binary one and a suitable number of binary zeroes (up to six) are appended to the data bits, and the completion tag (the least significant bit of the descriptor byte d_2) is set.

- Finally, r references to other cells follow. Each reference is normally represented by 32 bytes containing the sha256 hash of the referenced cell, computed as explained below in **1.1.4**.

In this way, the standard representation CellRepr(c) of a cell c with l data bits and r references is $2 + \lfloor l/8 \rfloor + \lceil l/8 \rceil + 32r$ bytes long.

1.1.4. The sha256 hash of a cell. The sha256 hash of a cell c is recursively defined as the sha256 of the standard representation CellRepr(c) of the cell in question:

$$\text{Hash}(c) := \text{sha256}(c) := \text{sha256}\big(\text{CellRepr}(c)\big) \tag{2}$$

Because cyclic cell references are not allowed (the relationships among all cells must constitute a directed acyclic graph, or DAG), the sha256 hash of a cell is always well-defined.

Furthermore, because sha256 is tacitly assumed to be collision-resistant, we assume that all the cells that we encounter are completely determined

by their hashes. In particular, the cell references of a cell c are completely determined by the hashes of the referenced cells, contained in the standard representation $\text{CellRepr}(c)$.

1.1.5. Exotic cells. Apart from the *ordinary* cells (also called *simple* or *data* cells) considered so far, cells of other types, called *exotic cells*, sometimes appear in the actual representations of TON Blockchain blocks and other data structures. Their representation is somewhat different; they are distinguished by having the first descriptor byte $d_1 \geq 5$ (cf. [4, 3.1]).

1.1.6. External reference cells. *(External) reference cells*, which contain the 32-byte $\text{sha256}(c)$ of a "true" data cell c instead of the data cell itself, are one example of exotic cells. These cells can be used in the serialization of a bag of cells corresponding to a TON Blockchain block in order to refer to data cells absent in the serialization of the block itself, but assumed to be present somewhere else (e.g., in the previous state of the blockchain).

1.1.7. Transparency of reference cells with respect to most operations. Most cell operations do not observe any reference cells or other "exotic" kinds of cells; they see only data cells, with any reference cell transparently replaced by the cell referred to. For example, when the *transparent* cell hash $\text{Hash}^{\flat}(c)$ is recursively computed, the hash of a reference cell is set to be equal to the hash of the cell referred to, not the hash of the standard representation of the reference cell.

1.1.8. Transparent hash and representation hash of a cell. In this way, $\text{sha256}^{\flat}(c) = \text{Hash}^{\flat}(c)$ is the *transparent hash* of a cell c (or the tree of cells rooted in c).

However, sometimes we need to reason about the exact representation of a tree of cells present in a block. To this end, a *representation hash* $\text{Hash}^{\sharp}(c)$ is defined, which is not transparent with respect to reference cells and other exotic types of cells. We often say that the representation hash of c is "the" hash of c, because it is the most frequently used hash of a cell.

1.1.9. Use of representation hashes for signatures. Signatures are an excellent example of the application of representation hashes. For instance:

- Validators sign the representation hash of a block, not just its transparent hash, because they need to certify that the block does contain the required data, not just some external references to them.

- When external messages are signed and sent by off-chain parties (e.g., human clients using an application to initiate blockchain transactions), if external references may be present in some of these messages, it is the representation hashes of the messages that must be signed.

1.1.10. Higher hashes of a cell. In addition to the transparent and representation hashes of a cell c, a sequence of *higher hashes* $\text{HASH}_i(c)$, $i = 1, 2, \ldots$ may be defined, which eventually stabilizes at $\text{HASH}_\infty(c)$. (More detail may be found in [4, 3.1].)

1.2 Principal components of a block and the blockchain state

This section briefly describes the principal components of a block and of the blockchain state, without delving too much into the details.

1.2.1. The Infinite Sharding Paradigm (ISP) applied to blockchain block and state. Recall that according to the Infinite Sharding Paradigm, each account can be considered as lying in its separate "accountchain", and the (virtual) blocks of these accountchains are then grouped into shardchain blocks for efficiency purposes. Specifically, the state of a shardchain consists, roughly speaking, of the states of all its "accountchains" (i.e., of all accounts assigned to it); similarly, a block of a shardchain essentially consists of a collection of virtual "blocks" for some accounts assigned to the shardchain.[5]

We can summarize this as follows:

$$\mathit{ShardState} \approx \mathit{Hashmap}(n, \mathit{AccountState}) \tag{3}$$

$$\mathit{ShardBlock} \approx \mathit{Hashmap}(n, \mathit{AccountBlock}) \tag{4}$$

where n is the bit length of the *account_id*, and $\mathit{Hashmap}(n, X)$ describes a partial map $\mathbf{2}^n \dashrightarrow X$ from bitstrings of length n into values of type X.

Recall that each shardchain—or, more precisely, each shardchain block[6]—corresponds to all accountchains that belong to the same "workchain" (i.e., have the same *workchain_id* $= w$) and have an *account_id* beginning with

[5] If there are no transactions related to an account, the corresponding virtual block is empty and is omitted in the shardchain block

[6] Recall that TON Blockchain supports *dynamic* sharding, so the shard configuration may change from block to block because of shard merge and split events. Therefore, we cannot simply say that each shardchain corresponds to a fixed set of accountchains.

the same binary prefix s, so that (w, s) completely determines a shard. Therefore, the above hashmaps must contain only keys beginning with prefix s.

We will see in a moment that the above description is only an approximation: the state and block of the shardchain need to contain some extra data that are not split according to the *account_id* as suggested by (3).

1.2.2. Split and non-split part of the shardchain block and state. A shardchain block and its state may each be classified into two distinct parts. The parts with the ISP-dictated form of (3) will be called the *split* parts of the block and its state, while the remainder will be called the *non-split* parts.

1.2.3. Interaction with other blocks and the outside world. Global and local consistency conditions. The non-split parts of the shardchain block and its state are mostly related to the interaction of this block with some other "neighboring" blocks. The global consistency conditions of the blockchain as a whole are reduced to internal consistency conditions of separate blocks by themselves as well as external local consistency conditions between certain blocks (cf. **1.3**).

Most of these local consistency conditions are related to message forwarding between different shardchains, transactions involving more than one shardchain, and message delivery guarantees. However, another group of local consistency conditions relates a block with its immediate antecessors and successors inside a shardchain; for instance, the initial state of a block usually must coincide with the final state of its immediate antecessor.[7]

1.2.4. Inbound and outbound messages of a block. The most important components of the non-split part of a shardchain block are the following:

- *InMsgDescr* — The description of all messages "imported" into this block (i.e., either processed by a transaction included in the block, or forwarded to an output queue, in the case of a transit message travelling along the path dictated by Hypercube Routing).
- *OutMsgDescr* — The description of all messages "exported" or "generated" by the block (i.e., either messages generated by a transaction included in the block, or transit messages with destination not belonging to the current shardchain, forwarded from *InMsgDescr*).

[7]This condition applies if there is exactly one immediate antecessor (i.e., if a shardchain merge event did not occur immediately before the block in question); otherwise, this condition becomes more convoluted.

1.2.5. Block header. Another non-split component of a shardchain block is the *block header*, which contains general information such as (w, s) (i.e., the *workchain_id* and the common binary prefix of all *account_id*s assigned to the current shardchain), the block's *sequence number* (defined to be the smallest non-negative integer larger than the sequence numbers of its predecessors), *logical time*, and *generation unixtime*. It also contains the hash of the immediate antecessor of the block (or of its two immediate antecessors in the case of a preceding shardchain merge event), the hashes of its initial and final states (i.e., of the states of the shardchain immediately before and immediately after processing the current block), and the hash of the most recent masterchain block known when the shardchain block was generated.

1.2.6. Validator signatures, signed and unsigned blocks. The block described so far is an *unsigned block*; it is generated in its entirety and considered as a whole by the validators. When the validators ultimately sign it, the *signed block* is created, consisting of the unsigned block along with a list of validator signatures (of a certain representation hash of the unsigned block, cf. **1.1.9**). This list of signatures is also a non-split component of the (signed) block; however, since it lies outside the unsigned block, it is somewhat different from the other data kept in a block.

1.2.7. Outbound message queue of a shardchain. Similarly, the most important non-split part of the shardchain state is *OutMsgQueue*, the outbound message queue. It contains *undelivered* messages included into *OutMsgDescr*, either by the last shardchain block leading to this state or by one of its antecessors.

Originally, each outbound message is included into *OutMsgQueue*; it is removed from the queue only after it has either been included into the *InMsgDescr* of a block of a "neighboring" shardchain (the next one with respect to Hypercube Routing), or has been delivered to (i.e., has appeared in the *InMsgDescr* of) its ultimate destination shardchain via Instant Hypercube Routing. In both cases, the *reason* for the removal of a message from the *OutMsgQueue* is made explicit in the *OutMsgDescr* of the block in which such a state transformation has occurred.

1.2.8. Layout of *InMsgDescr*, *OutMsgDescr* and *OutMsgQueue*. All of the most important non-split shardchain data structures related to messages are organized as *hashmaps* or *dictionaries* (implemented by means of Patricia trees serialized into a tree of cells as described in [4, 3.3]), with

the following keys:

- The inbound message description *InMsgDescr* uses the 256-bit message hash as a key.
- The outbound message description *OutMsgDescr* uses the 256-bit message hash as a key.
- The outbound message queue *OutMsgQueue* uses the 352-bit concatenation of the 32-bit destination *workchain_id*, the first 64 bits of destination address *account_id*, and the 256-bit message hash as a key.

1.2.9. The split part of the block: transaction chains. The split part of a shardchain block consists of a hashmap mapping some of the accounts assigned to the shardchain to "virtual accountchain blocks" *AccountBlock*, cf. (3). Such a virtual accountchain block consists of a sequential list of *transactions* related to that account.

1.2.10. Transaction description. Each transaction is described in the block by an instance of the *Transaction* type, which contains in particular the following information:

- A reference to exactly one *inbound message* (which must be present in *InMsgDescr* as well) that has been *processed* by the transaction.
- References to several (maybe zero) *outbound messages* (also present in *OutMsgDescr* and most likely included in *OutMsgQueue*) that have been *generated* by the transaction.

The transaction consists of an invocation of TVM (cf. [4]) with the code of the smart contract corresponding to the account in question loaded into the virtual machine, and with the data root cell of the smart contract loaded into the virtual machine's register `c4`. The inbound message itself is passed in the stack as an argument to the smart contract's `main()` function, along with some other important data, such as the amount of TON Grams and other defined currencies attached to the message, the sender account address, the current balance of the smart contract, and so on.

In addition to the information listed above, a *Transaction* instance also contains the original and final states of the account (i.e., of the smart contract), as well as some of the TVM running statistics (gas consumed, gas price, instructions performed, cells created/destroyed, virtual machine termination code, etc.).

1.2.11. The split part of the shardchain state: account states. Recall that, according to (3), the split part of the shardchain state consists of a hashmap mapping each "defined" account identifier (belonging to the shardchain in question) to the *state* of the corresponding account, given by an instance of the *AccountState* type.

1.2.12. Account state. The account state itself approximately consists of the following data:

- Its *balance* in Grams and (optionally) in some other defined cryptocurrencies/tokens.
- The *smart-contract code*, or the hash of the smart-contract code if it will be provided (uploaded) later by a separate message.
- The persistent *smart-contract data*, which can be empty for simple smart contracts. It is a tree of cells, the root of which is loaded into register `c4` during smart-contract execution.
- Its *storage usage statistics*, including the number of cells and bytes kept in the persistent storage of the smart contract (i.e., inside the blockchain state) and the last time a storage usage payment was exacted from this account.
- An optional *formal interface description* (intended for smart contracts) and/or *user public information* (intended mostly for human users and organizations).

Notice that there is no distinction between "smart contract" and "account" in the TON Blockchain. Instead, "simple" or "wallet" accounts, typically employed by human users and their cryptocurrency wallet applications for simple cryptocurrency transfers, are just simple smart contracts with standard (shared) code and with persistent data consisting of the public key of the wallet (or several public keys in the case of a multi-signature wallet; cf. **1.7.6** for more detail).

1.2.13. Masterchain blocks. In addition to shardchain blocks and their states, the TON Blockchain contains *masterchain blocks* and the *masterchain state* (also called the *global state*). The masterchain blocks and state are quite similar to the shardchain blocks and state considered so far, with some notable differences:

- The masterchain cannot be split or merged, so a masterchain block usually has exactly one immediate antecessor. The sole exception is the "masterchain block zero", distinguished by having a sequence number equal to zero; it has no antecessors at all, and contains the initial configuration of the whole TON Blockchain (e.g., the original set of validators).

- The masterchain blocks contain another important non-split structure: *ShardHashes*, a binary tree with a list of all defined shardchains along with the hashes of the latest block inside each of the listed shardchains. It is the inclusion of a shardchain block into this structure that makes a shardchain block "canonical", and enables other shardchains' blocks to refer to data (e.g., outbound messages) contained in the shardchain block.

- The state of the masterchain contains global configuration parameters of the whole TON Blockchain, such as the minimum and maximum gas prices, the supported versions of TVM, the minimum stake for the validator candidates, the list of alternative cryptocurrencies supported in addition to Grams, the total amount of Grams issued so far, and the current set of validators responsible for creating and signing new blocks, along with their public keys.

- The state of the masterchain also contains the code of the smart contracts used to elect the subsequent sets of validators and to modify the global configuration parameters. The code of these smart contracts itself is a part of the global configuration parameters and can be modified accordingly. In this respect, this code (along with the current values of these parameters) functions like a "constitution" for the TON Blockchain. It is initially established in masterchain block zero.

- There are no transit messages through the masterchain: each inbound message must have a destination inside the masterchain, and each outbound message must have a source inside the masterchain.

1.3 Consistency conditions

In addition to the data structures contained in the block and in the blockchain state, which are serialized into bags of cells according to certain TL-B schemes

explained in detail later (cf. Chapters **3**–**5**), an important component of the blockchain layout is the *consistency conditions* between data kept inside one or in different blocks (as mentioned in **1.2.3**). This section describes in detail the function of consistency conditions in the blockchain.

1.3.1. Expressing consistency conditions. In principle, dependent data types (such as those used in TL-B) could be used not only to describe the serialization of block data, but also to express conditions imposed on the components of such data types. (For instance, one could define data type *OrderedIntPair*, with pairs of integers (x, y), such that $x < y$, as values.) However, TL-B currently is not expressive enough to encode all the consistency conditions we need, so we opt for a semi-formalized approach in this text. In the future, we may present a subsequent complete formalization in a suitable proof assistant such as Coq.

1.3.2. Importance of consistency conditions. The consistency conditions ultimately are at least as important as the "unrestricted" data structures on which they are imposed, especially in the blockchain context. For instance, the consistency conditions ensure that the state of an account does not change between blocks, and that it can change within a block only as a result of a transaction. In this way, the consistency conditions ensure the safe storage of cryptocurrency balances and other information inside the blockchain.

1.3.3. Kinds of consistency conditions. There are several kinds of consistency conditions imposed on the TON Blockchain:

- *Global conditions* — Express the invariants throughout the entire TON Blockchain. For instance, the *message delivery guarantees*, which assert that each message generated must be delivered to its destination account and delivered exactly once, are part of the global conditions.

- *Internal (local) conditions* — Express the conditions imposed on the data kept inside one block. For example, each transaction included in the block (i.e., present in the transaction list of some account) processes exactly one inbound message; this inbound message must be listed in the *InMsgDescr* structure of the block as well.

- *External (local) conditions* — Express the conditions imposed on the data of different blocks, usually belonging to the same or to neighbor-

ing shardchains (with respect to Hypercube Routing). Therefore, the external conditions come in several flavors:

- *Antecessor/successor conditions* — Express the conditions imposed on the data of some block and of its immediate antecessor or (in the case of a preceding shardchain merge event) two immediate antecessors. The most important of these conditions is the one stating that the initial state for a shardchain block must coincide with final shardchain state of the immediate antecessor block, provided no shardchain split/merge event happened in between.
- *Masterchain/shardchain conditions* — Express the conditions imposed on a shardchain block and on the masterchain block that refers to it in its *ShardHashes* list or is referred to in the header of the shardchain block.
- *Neighbor (block) conditions* — Express the relations between the blocks of neighboring shardchains with respect to Hypercube Routing. The most important of these conditions express the relation between the *InMsgDescr* of a block and the *OutMsgQueue* of the state of a neighboring block.

1.3.4. Decomposition of global and local conditions into simpler local conditions. The *global* consistency conditions, such as the message delivery guarantees, are truly necessary for the blockchain to work properly; however, they are hard to enforce and verify directly. Therefore, we instead introduce a lot of simpler *local* consistency conditions, which are easier to enforce and verify since they involve only one block, or perhaps two adjacent blocks. These local conditions are chosen in such a fashion that the desired global conditions are logical consequences of (the conjunction of) all the local conditions. In this respect, we say that the global conditions have been "decomposed" into simpler local conditions.

Sometimes a local condition still turns out to be too cumbersome to enforce or verify. In that case it is decomposed further, into even simpler local conditions.

1.3.5. Decomposition may require additional data structures and additional internal consistency conditions. The decomposition of a condition into simpler local consistency conditions sometimes requires the introduction of additional data structures. For example, the *InMsgDescr*

explicitly lists all inbound messages processed in a block, even if this list might have been obtained by scanning the list of all the transactions present in the block. However, *InMsgDescr* greatly simplifies the neighbor conditions related to message forwarding and routing, which ultimately add up to the global message delivery guarantees.

Notice that the introduction of such additional data structures is a sort of "database denormalization" (i.e., it leads to some redundancy, or to some data being present more than once), and therefore more internal consistency conditions need to be imposed (e.g., if some data are now present in two copies, we must require that these two copies coincide). For instance, once we introduce *InMsgDescr* to facilitate message forwarding between shardchains, we need to introduce internal consistency conditions relating *InMsgDescr* to the transaction list of the same block.

1.3.6. Correct serialization conditions. Apart from the high-level internal consistency conditions, which treat the contents of a block as a value of an abstract data type, there are some lower-level internal consistency conditions, called "(correct) serialization conditions", which ensure that the tree of cells present in the block is indeed a valid serialization of a value of the expected abstract data type. Such serialization conditions can be automatically generated from the TL-B scheme describing the abstract data type and its serialization into a tree of cells.

Notice that the serialization conditions are a set of mutually recursive predicates on cells or cell slices. For example, if a value of type A consists of a 32-bit magic number m_A, a 64-bit integer l, and two references to cells containing values of types B and C, respectively, then the correct serialization condition for values of type A will require a cell or a cell slice to contain exactly 96 data bits and two cell references r_1 and r_2, with the additional requirements that the first 32 data bits contain m_A, and the two cells referred to by r_1 and r_2 satisfy the serialization conditions for values of types B and C, respectively.

1.3.7. Constructive elimination of existence quantifiers. The local conditions one might want to impose sometimes are *non-constructible*, meaning that they do not necessarily contain an explanation of why they are true. A typical example of such a condition C is given by

$$C :\equiv \forall_{(x:X)} \exists_{(y:Y)} A(x,y) \quad , \tag{5}$$

"for any x from X, there is a y from Y such that condition $A(x, y)$ holds". Even if we know C to be true, we do not have a way of quickly finding a $y : Y$, such that $A(x, y)$, for a given $x : X$. As a consequence, the verification of C may be quite time-consuming.

In order to simplify the verification of local conditions, they are made *constructible* (i.e., verifiable in bounded time) by adding some *witness* data structures. For instance, condition C of (5) may be transformed by adding a new data structure $f : X \to Y$ (a map f from X to Y) and imposing the following condition C' instead:

$$C' :\equiv \forall_{(x:X)} A\big(x, f(x)\big) \quad . \tag{6}$$

Of course, the "witness" value $f(x) : Y$ may be included inside the (modified) data type X instead of being kept in a separate table f.

1.3.8. Example: consistency condition for *InMsgDescr*. For instance, the consistency condition between $X := \mathit{InMsgDescr}$, the list of all inbound messages processed in a block, and $Y := \mathit{Transactions}$, the list of all transactions present in a block, is of the above sort: "For any input message x present in *InMsgDescr*, a transaction y must be present in the block such that y processes x".[8] The procedure of $\exists$-elimination described in **1.3.7** leads us to introduce an additional field in the inbound message descriptors of *InMsgDescr*, containing a reference to the transaction in which the message is actually processed.

1.3.9. Constructive elimination of logical disjunctions. Similarly to the transformation described in **1.3.7**, condition

$$D :\equiv \forall_{(x:X)} \big(A_1(x) \vee A_2(x)\big) \quad , \tag{7}$$

"for all x from X, at least one of $A_1(x)$ and $A_2(x)$ holds", may be transformed into a function $i : X \to \mathbf{2} = \{1, 2\}$ and a new condition

$$D' :\equiv \forall_{(x:X)} A_{i(x)}(x) \tag{8}$$

This is a special case of the existential quantifier elimination considered before for $Y = \mathbf{2} = \{1, 2\}$. It may be useful when $A_1(x)$ and $A_2(x)$ are complicated conditions that cannot be verified quickly, so that it is useful to know in advance which of them is in fact true.

[8] This example is a bit simplified since it does not take into account the presence of transit messages in *InMsgDescr*, which are not processed by any explicit transaction.

For instance, *InMsgDescr*, as considered in **1.3.8**, can contain both messages processed in the block and transit messages. We might introduce a field in the inbound message description to indicate whether the message is transit or not, and, in the latter case, include a witness field for the transaction processing the message.

1.3.10. Constructivization of conditions. This process of eliminating the non-constructible logical binders $\exists$ (existence quantifier) and (sometimes) $\vee$ (logical disjunction) by introducing additional data structures and fields—that is, the process of making a condition constructible—will be called *constructivization*. If taken to its theoretical limit, this process leads to logical formulas containing only universal quantifiers and logical conjunctions, at the expense of adding some witness fields into certain data structures.

1.3.11. Validity conditions for a block. Ultimately, all of the internal conditions for a block, along with the local antecessor and neighbor conditions involving this block and another previously generated block, constitute the *validity conditions* for a shardchain or masterchain block. A block is *valid* if it satisfies the validity conditions. It is the responsibility of validators to generate valid blocks, as well as check the validity of blocks generated by other validators.

1.3.12. Witnesses of the invalidity of a block. If a block does not satisfy all of the validity conditions $C_1, \ldots, C_n$ (i.e., the conjunction $V :\equiv \bigwedge_i C_i$ of the validity conditions), it is *invalid*. This means that it satisfies the "invalidity condition" $\neg V = \bigvee_i \neg C_i$. If all of the C_i—and hence, also V—have been "constructivized" in the sense described in **1.3.10**, so that they contain only logical conjunctions and universal quantifiers (and simple atomic propositions), then $\neg V$ contains only logical disjunctions and existential quantifiers. Then a constructivization of $\neg V$ may be defined, which would involve an *invalidity witness*, starting with an index i of the specific validity condition C_i which fails.

Such invalidity witnesses may also be serialized and presented to other validators or committed into the masterchain to prove that a specific block or block candidate is in fact invalid. Therefore, the construction and serialization of invalidity witnesses is an important part of a Proof-of-Stake (PoS) blockchain design.[9]

[9]It is interesting to note that this part of the work can be done almost automatically.

1.3.13. Minimizing the size of witnesses. An important consideration for the design of the local conditions, their decomposition into simpler conditions, and their constructivization is to make the verification of each condition as simple as possible. However, another requirement is that we should minimize the size of witnesses both for a condition (so that block size does not grow too much during the constructivization process) and for its negation (so that the invalidity proofs have bounded size, which simplifies their verification, transmission, and inclusion into the masterchain). These two design principles are sometimes at odds, and a compromise must be then sought.

1.3.14. Minimizing the size of Merkle proofs. The consistency conditions are originally intended to be processed by a party who already has all the relevant data (e.g., all the blocks mentioned in the condition). On some occasions, however, they must be verified by a party who does not have all the blocks in question, but knows only their hashes. For example, suppose that a block invalidity proof were augmented by the signature of a validator that had signed an invalid block (and therefore would have to be punished). In this case, the signature would contain only the hash of the wrongly signed block; the block itself would have to be recovered from a different place before verifying the block invalidity proof.

A compromise between providing only the hash of the supposedly invalid block and providing the entire invalid block along with the invalidity witness is to augment the invalidity witness by a Merkle proof starting from the hash of the block (i.e., of the root cell of the block). Such a proof would include all the cells referred to in the invalidity witness, along with all the cells on the paths from these cells to the root cells and the hashes of their siblings. Then an invalidity proof becomes self-contained enough to provide sufficient justification on its own for punishing a validator. For example, the invalidity proof suggested above might be presented to a smart contract residing in the masterchain that punishes the validators for incorrect behavior.

Since such an invalidity proof must be augmented by a Merkle proof, it makes sense to write the consistency conditions so that the Merkle proofs for their negations would be as small as possible. In particular, each individual condition must be as "local" as possible (i.e., involve a minimal number of cells). This also optimizes the verification time of the invalidity proof.

1.3.15. Collated data for the external conditions. When a validator suggests an unsigned block to the other validators of a shardchain, these other validators must check the validity of this block candidate—i.e., verify

that it satisfies all of the internal and external local consistency conditions. While the internal conditions do not require any extra data in addition to the block candidate itself, the external conditions need some other blocks, or at least some information out of those blocks. Such additional information may be extracted from those blocks, along with all cells on the paths from the cells containing the required additional information to the root cell of the corresponding blocks and the hashes of the siblings of the cells on these paths, to present a Merkle proof that can be processed without knowledge of the referred blocks themselves.

This additional information, called *collated data*, is serialized as a bag of cells and presented by the validator along with the unsigned block candidate itself. The block candidate along with the collated data is called a *collated block*.

1.3.16. Conditions for a collated block. The *external* consistency conditions for a block candidate are thus (automatically) transformed into *internal* consistency conditions for a collated block, which greatly simplifies and speeds up their verification by the other validators. However, some data—such as the final state of the immediate antecessor of the block being validated—is not collated. Instead, all validators are supposed to keep a local copy of this data.

1.3.17. Representation conditions and representation hashes. Notice that once Merkle proofs are included into a collated block, the consistency conditions must take into account which data (i.e., which cells) are actually present in the collated block, and not just referred to by their hashes. This leads to a new group of conditions, called *representation conditions*, which must be able to distinguish an external cell reference (usually represented by its 256-bit hash) from the cell itself. A validator can be punished for suggesting a collated block that does not contain all of the expected collated data inside, even if the block candidate itself is valid.

This also leads to the utilization of *representation hashes* instead of *transparent hashes* for collated blocks.

1.3.18. Verification in the absence of the collated data. Notice that a block must still be verifiable in the absence of the collated data; otherwise, no party except the validators would be able to check a previously committed block by its own means. In particular, witnesses cannot be included into the collated data: they must reside in the block itself. The collated data

must contain only some portions of neighboring blocks referred to in the principal block along with suitable Merkle proofs, which can be reconstructed by anybody who has the referenced blocks themselves.

1.3.19. Inclusion of Merkle proofs in the block itself. Notice that on some occasions Merkle proofs must be embedded into the block itself, and not just into collated data. For instance:

- During Instant Hypercube Routing (IHR), a message may be included directly into the *InMsgDescr* of a block of the destination shardchain, without travelling all the way along the edges of the hypercube. In this case, a Merkle proof of the existence of the message in the *OutMsgDescr* of a block of the originating shardchain must be included into *InMsgDescr* along with the message itself.

- An invalidity proof, or another proof of validator misbehavior, may be committed into the masterchain by including it in the body of a message sent to a special smart contract. In this case, the invalidity proof must include some cells along with a Merkle proof, which must therefore be contained in a message body.

- Similarly, a smart contract defining a payment channel, or another kind of side-chain, may accept finalization messages or misbehavior proof messages that contain suitable Merkle proofs.

- The final state of a shardchain is not included into a shardchain block. Instead, only the cells that have been modified are included; those cells that are inherited from the old state are referred to by their hashes, along with suitable Merkle proofs consisting of the cells on the path from the root of the old state to the cells of the old state referred to.

1.3.20. Provisions for handling incomplete data. As we have seen, it is necessary to include incomplete data and Merkle proofs into the body of a block, into the body of some messages contained in a block, and into the state. This necessity is reflected by some extra representation conditions, as well as provisions for the messages (and by extension, the cell trees processed by TVM) to contain incomplete data (external cell references and Merkle proofs). In most cases, such external cell references contain only the 256-bit SHA256 hash of a cell along with a flag; if a smart contract attempts

to inspect the contents of such a cell by a `CTOS` primitive (e.g., for deserialization), an exception is triggered. However, an external reference to such a cell can be stored into the smart contract's persistent storage, and both the transparent and the representation hashes of such a cell can be computed.

1.4 Logical time and logical time intervals

This section takes a closer look at so-called *logical time*, extensively used in the TON Blockchain for message forwarding and message delivery guarantees, among other purposes.

1.4.1. Logical time. A component of the TON Blockchain that also plays an important role in message delivery is the *logical time*, usually denoted by $\textsc{Lt}$. It is a non-negative 64-bit integer, assigned to certain events roughly as follows:

> If an event e logically depends on events $e_1, \ldots, e_n$, then $\textsc{Lt}(e)$ is the smallest non-negative integer greater than all $\textsc{Lt}(e_i)$.

In particular, if $n = 0$ (i.e., if e does not depend on any prior events), then $\textsc{Lt}(e) = 0$.

1.4.2. A relaxed variant of logical time. On some occasions we relax the definition of logical time, requesting only that

$$\textsc{Lt}(e) > \textsc{Lt}(e') \quad \text{whenever } e \succ e' \text{ (i.e., } e \text{ logically depends on } e'), \tag{9}$$

without insisting that $\textsc{Lt}(e)$ be the smallest non-negative integer with this property. In such cases we can speak about *relaxed* logical time, as opposed to the *strict* logical time defined above (cf. **1.4.1**). Notice, however, that the condition (9) is a fundamental property of logical time and cannot be relaxed further.

1.4.3. Logical time intervals. It makes sense to assign to some events or collections of events C an *interval* of logical times $\textsc{Lt}^{\bullet}(C) = [\textsc{Lt}^{-}(C), \textsc{Lt}^{+}(C))$, meaning that the collection of events C took place in the specified "interval" of logical times, where $\textsc{Lt}^{-}(C) < \textsc{Lt}^{+}(C)$ are some integers (64-bit integers in practice). In this case, we can say that C *begins* at logical time $\textsc{Lt}^{-}(C)$, and *ends* at logical time $\textsc{Lt}^{+}(C)$.

By default, we assume $\textsc{Lt}^{+}(e) = \textsc{Lt}(e)+1$ and $\textsc{Lt}^{-}(e) = \textsc{Lt}(e)$ for simple or "atomic" events, assuming that they last exactly one unit of logical time.

In general, if we have a single value $\mathrm{LT}(C)$ as well as logical time interval $\mathrm{LT}^\bullet(C) = [\mathrm{LT}^-(C), \mathrm{LT}^+(C))$, we always require that

$$\mathrm{LT}(C) \in [\mathrm{LT}^-(C), \mathrm{LT}^+(C)) \tag{10}$$

or, equivalently,

$$\mathrm{LT}^-(C) \le \mathrm{LT}(C) < \mathrm{LT}^+(C) \tag{11}$$

In most cases, we choose $\mathrm{LT}(C) = \mathrm{LT}^-(C)$.

1.4.4. Requirements for logical time intervals. The three principal requirements for logical time intervals are:

- $0 \le \mathrm{LT}^-(C) < \mathrm{LT}^+(C)$ are non-negative integers for any collection of events C.
- If $e' \prec e$ (i.e., if an atomic event e logically depends on another atomic event e'), then $\mathrm{LT}^\bullet(e') < \mathrm{LT}^\bullet(e)$ (i.e., $\mathrm{LT}^+(e') \le \mathrm{LT}^-(e)$).
- If $C \supset D$ (i.e., if a collection of events C contains another collection of events D), then $\mathrm{LT}^\bullet(C) \supset \mathrm{LT}^\bullet(D)$, i.e.,

$$\mathrm{LT}^-(C) \le \mathrm{LT}^-(D) < \mathrm{LT}^+(D) \le \mathrm{LT}^+(C) \tag{12}$$

  In particular, if C consists of atomic events $e_1, \ldots, e_n$, then $\mathrm{LT}^-(C) \le \inf_i \mathrm{LT}^-(e_i) \le \inf_i \mathrm{LT}(e_i)$ and $\mathrm{LT}^+(C) \ge \sup_i \mathrm{LT}^+(e_i) \ge 1 + \sup_i \mathrm{LT}(e_i)$.

1.4.5. Strict, or minimal, logical time intervals. One can assign to any finite collection of atomic events $E = \{e\}$ related by a causality relation (partial order) $\prec$, and all subsets $C \subset E$, *minimal* logical time intervals. That is, among all assignments of logical time intervals satisfying the conditions listed in **1.4.4**, we choose the one having all $\mathrm{LT}^+(C) - \mathrm{LT}^-(C)$ as small as possible, and if several assignments with this property exist, we choose the one that has the minimum $\mathrm{LT}^-(C)$ as well.

Such an assignment can be achieved by first assigning logical time $\mathrm{LT}(e)$ to all atomic events $e \in E$ as described in **1.4.1**, then setting $\mathrm{LT}^-(C) := \inf_{e \in C} \mathrm{LT}(e)$ and $\mathrm{LT}^+(C) := 1 + \sup_{e \in C} \mathrm{LT}(e)$ for any $C \subset E$.

In most cases when we need to assign logical time intervals, we use the minimal logical time intervals just described.

1.4.6. Logical time in the TON Blockchain. The TON Blockchain assigns logical time and logical time intervals to several of its components.

For instance, each outbound message created in a transaction is assigned its *logical creation time*; for this purpose, the creation of an outbound message is considered an atomic event, logically dependent on the previous message created by the same transaction, as well as on the previous transaction of the same account, on the inbound message processed by the same transaction, and on all events contained in the blocks referred to by hashes contained in the block with the same transaction. As a consequence, *outbound messages created by the same smart contract have strictly increasing logical creation times.* The transaction itself is considered a collection of atomic events, and is assigned a logical time interval (cf. **4.2.1** for a more precise description).

Each block is a collection of transaction and message creation events, so it is assigned a logical time interval, explicitly mentioned in the header of the block.

1.5 Total blockchain state

This section discusses the total state of the TON Blockchain, as well as the states of separate shardchains and the masterchain. For example, the precise definition of the state of the neighboring shardchains becomes crucial for correctly formalizing the consistency condition asserting that the validators for a shardchain must import the oldest messages from the union of *OutMsgQueue*s taken from the states of all neighboring shardchains (cf. **2.2.5**).

1.5.1. Total state defined by a masterchain block. Every masterchain block contains a list of all currently active shards and of the latest blocks for each of them. In this respect, *every masterchain block defines the corresponding total state of the TON Blockchain, since it fixes the state of every shardchain, and of the masterchain as well.*

An important requirement imposed on this list of the latest blocks for all shardchain blocks is that, if a masterchain block B lists S as the latest block of some shardchain, and a newer masterchain block B', with B as one of its antecessors, lists S' as the latest block of the same shardchain, then S must be one of the antecessors of S'.[10] This condition makes the total state of the

[10]In order to express this condition correctly in the presence of dynamic sharding, one should fix some account ξ, and consider the latest blocks S and S' of the shardchains containing ξ in the shard configurations of both B and B', since the shards containing ξ

TON blockchain defined by a subsequent masterchain block B' compatible with the total state defined by a previous block B.

1.5.2. Total state defined to by a shardchain block. Every shardchain block contains the hash of the most recent masterchain block in its header. Consequently, all the blocks referred to in that masterchain block, along with their antecessors, are considered "known" or "visible" to the shardchain block, and no other blocks are visible to it, with the sole exception of its antecessors inside its proper shardchain.

In particular, when we say that a block *must* import in its *InMsgDescr* the messages from the *OutMsgQueue* of the states of all neighboring shardchains, it means that precisely the blocks of other shardchains visible to that block must be taken into account, and at the same time the block cannot contain messages from "invisible" blocks, even if they are otherwise correct.

1.6 Configurable parameters and smart contracts

Recall that the TON Blockchain has several so-called "configurable parameters" (cf. [3]), which are either certain values or certain smart contracts residing in the masterchain. This section discusses the storage of and access to these configurable parameters.

1.6.1. Examples of configurable parameters. The properties of the blockchain controlled by configurable parameters include:

- The minimum stake for validators.
- The maximum size of the group of elected validators.
- The maximum number of blocks for which the same group of validators are responsible.
- The validator election process.
- The validator punishing process.
- The currently active and the next elected set of validators.

might be different in B and B'.

- The process of changing configurable parameters, and the address of the smart contract γ responsible for holding the values of the configurable parameters and for modifying their values.

1.6.2. Location of the values of configurable parameters. The configurable parameters are kept in the persistent data of a special configuration smart contract γ residing in the masterchain of the TON Blockchain. More precisely, the first reference of the root cell of the persistent data of that smart contract is a dictionary mapping 64-bit keys (parameter numbers) to the values of the corresponding parameters; each value is serialized into a cell slice according to the type of that value. If a value is a "smart contract" (necessarily residing in the masterchain), its 256-bit account address is used instead.

1.6.3. Quick access through the header of masterchain blocks. To simplify access to the current values of configurable parameters, and to shorten the Merkle proofs containing references to them, the header of each masterchain block contains the address of smart contract γ. It also contains a direct cell reference to the dictionary containing all values of configurable parameters, which lies in the persistent data of γ. Additional consistency conditions ensure that this reference coincides with the one obtained by inspecting the final state of smart contract γ.

1.6.4. Getting values of configurable parameters by get methods. The configuration smart contract γ provides access to some of configurable parameters by means of "get methods". These special methods of the smart contract do not change its state, but instead return required data in the TVM stack.

1.6.5. Getting values of configurable parameters by get messages. Similarly, the configuration smart contract γ may define some "ordinary" methods (i.e., special inbound messages) to request the values of certain configuration parameters, which will be sent in the outbound messages generated by the transaction processing such an inbound message. This may be useful for some other fundamental smart contracts that need to know the values of certain configuration parameters.

1.6.6. Values obtained by get methods may be different from those obtained through the block header. Notice that the state of the configuration smart contract γ, including the values of configurable parameters,

may change several times inside a masterchain block, if there are several transactions processed by γ in that block. As a consequence, the values obtained by invoking get methods of γ, or sending get messages to γ, may be different from those obtained by inspecting the reference in the block header (cf. **1.6.3**), which refers to the *final* state of the configurable parameters in the block.

1.6.7. Changing the values of configurable parameters. The procedure for changing the values of configurable parameters is defined in the code of smart contract γ. For most configurable parameters, called *ordinary*, any validator may suggest a new value by sending a special message with the number of the parameter and its proposed value to γ. If the suggested value is valid, further voting messages from the validators are collected by the smart contract, and if more than two-thirds each of the current and next sets of validators support the proposal, the value is changed.

Some parameters, such as the current set of validators, cannot be changed in this way. Instead, the current configuration contains a parameter with the address of smart contract ν responsible for electing the next set of validators, and smart contract γ accepts messages only from this smart contract ν to modify the value of the configuration parameter containing the current set of validators.

1.6.8. Changing the validator election procedure. If the validator election procedure ever needs to be changed, this can be accomplished by first committing a new validator election smart contract into the masterchain, and then changing the ordinary configurable parameter containing the address ν of the validator election smart contract. This will require two-thirds of the validators to accept the proposal in a vote as described above in **1.6.7**.

1.6.9. Changing the procedure of changing configurable parameters. Similarly, the address of the configuration smart contract itself is a configurable parameter and may be changed in this fashion. In this way, most fundamental parameters and smart contracts of the TON Blockchain may be modified in any direction agreed upon by the qualified majority of the validators.

1.6.10. Initial values of the configurable parameters. The initial values of most configurable parameters appear in block zero of the masterchain as part of the masterchain's initial state, which is explicitly present with no

omissions in this block. The code of all fundamental smart contracts is also present in the initial state. In this way, the original "constitution" and configuration of the TON Blockchain, including the original set of validators, is made explicit in block zero.

1.7 New smart contracts and their addresses

This section discusses the creation and initialization of new smart contracts—in particular, the origin of their initial code, persistent data, and balance. It also discusses the assignment of account addresses to new smart contracts.

1.7.1. Description valid only for masterchain and basic workchain. The mechanisms for creating new smart contracts and assigning their addresses described in this section are valid only for the basic workchain and the masterchain. Other workchains may define their own mechanisms for dealing with these problems.

1.7.2. Transferring cryptocurrency to uninitialized accounts. First of all, *it is possible to send messages, including value-bearing messages, to previously unmentioned accounts.* If an inbound message arrives at a shardchain with a destination address η corresponding to an undefined account, it is processed by a transaction as if the code of the smart contract were empty (i.e., consisting of an implicit `RET`). If the message is value-bearing, this leads to the creation of an "uninitialized account", which may have a non-zero balance (if value-bearing messages have been sent to it),[11] but has no code and no data. Because even an uninitialized account occupies some persistent storage (needed to hold its balance), some small persistent-storage payments will be exacted from time to time from the account's balance, until it becomes negative.

1.7.3. Initializing smart contracts by constructor messages. An account, or smart contract, is created by sending a special *constructor message* M to its address η. The body of such a message contains the tree of cells with the initial code of the smart contract (which may be replaced by its hash in some situations), and the initial data of the smart contract (maybe empty; it can be replaced by its hash). The hash of the code and of the data

[11] Value-bearing messages with the `bounce` flag set will not be accepted by an uninitialized account, but will be "bounced" back.

contained in the constructor message must coincide with the address η of the smart contract; otherwise, it is rejected.

After the code and data of the smart contract are initialized from the body of the constructor message, the remainder of the constructor message is processed by a transaction (the *creating transaction* for smart contract η) by invoking TVM in a manner similar to that used for processing ordinary inbound messages.

1.7.4. Initial balance of a smart contract. Notice that the constructor message usually must bear some value, which will be transferred to the balance of the newly-created smart contract; otherwise, the new smart contract would have a balance of zero and would not be able to pay for storing its code and data in the blockchain. The minimum balance required from a newly-created smart contract is a linear (more precisely, affine) function of the storage it uses. The coefficients of this function may depend on the workchain; in particular, they are higher in the masterchain than in the basic workchain.

1.7.5. Creating smart contracts by external constructor messages. In some cases, it is necessary to create a smart contract by a constructor message that cannot bear any value—for instance, by a constructor message "from nowhere" (an external inbound message). Then one should first transfer a sufficient amount of funds to the uninitialized smart contract as explained in **1.7.2**, and only then send a constructor message "from nowhere".

1.7.6. Example: creating a cryptocurrency wallet smart contract. An example of the above situation is provided by cryptocurrency wallet applications for human users, which must create a special wallet smart contract in the blockchain in which to keep the user's funds. This can be achieved as follows:

- The cryptocurrency wallet application generates a new cryptographic public/private key pair (typically for Ed25519 elliptic curve cryptography, supported by special TVM primitives) for signing the user's future transactions.

- The cryptocurrency wallet application knows the code of the smart contract to be created (which typically is the same for all users), as well as the data, which typically consists of the public key of the wallet

(or of its hash) and is generated at the very beginning. The hash of this information is the address ξ of the wallet smart contract to be created.

- The wallet application may display the user's address ξ, and the user may start to receive funds to her uninitialized account ξ—for example, by buying some cryptocurrency at an exchange, or by asking a friend to transfer a small sum.
- The wallet application can inspect the shardchain containing account ξ (in the case of a basic workchain account) or the masterchain (in the case of a masterchain account), either by itself or using a blockchain explorer, and check the balance of ξ.
- If the balance is sufficient, the wallet application may create and sign (with the user's private key) the constructor message ("from nowhere"), and submit it for inclusion to the validators or the collators for the corresponding blockchain.
- Once the constructor message is included into a block of the blockchain and processed by a transaction, the wallet smart contract is finally created.
- When the user wants to transfer some funds to some other user or smart contract η, or wants to send a value-bearing message to η, she uses her wallet application to create the message m that she wants her wallet smart contract ξ to send to η, envelope m into a special "message from nowhere" m' with destination ξ, and sign m' with her private key. Some provisions against replay attacks must be made, as explained in **2.2.1**.
- The wallet smart contract receives message m' and checks the validity of the signature with the aid of the public key stored in its persistent data. If the signature is correct, it extracts embedded message m from m' and sends it to its intended destination η, with the indicated amount of funds attached to it.
- If the user does not need to immediately start transferring funds, but only wants to passively receive some funds, she may keep her account uninitialized as long as she wants (provided the persistent storage payments do not lead to the exhaustion of its balance), thus minimizing the storage profile and persistent storage payments of the account.

- Notice that the wallet application may create for the human user the illusion that the funds are kept in the application itself, and provide an interface to transfer funds or send arbitrary messages "directly" from the user's account ξ. In reality, all these operations will be performed by the user's wallet smart contract, which effectively acts as a proxy for such requests. We see that a cryptocurrency wallet is a simple example of a *mixed* application, having an on-chain part (the wallet smart contract, used as a proxy for outbound messages) and an off-chain part (the external wallet application running on a user's device and keeping the private account key).

Of course, this is just one way of dealing with the simplest user wallet smart contracts. One can create multi-signature wallet smart contracts, or create a shared wallet with internal balances kept inside it for each of its individual users, and so on.

1.7.7. Smart contracts may be created by other smart contracts. Notice that a smart contract may generate and send a constructor message while processing any transaction. In this way, smart contracts may automatically create new smart contracts, if they need to, without any human intervention.

1.7.8. Smart contracts may be created by wallet smart contracts. On the other hand, a user may compile the code for her new smart contract ν, generate the corresponding constructor message m, and use the wallet application to force her wallet smart contract ξ to send message m to ν with an adequate amount of funds, thus creating the new smart contract ν.

1.8 Modification and removal of smart contracts

This section explains how the code and state of a smart contract may be changed, and how and when a smart contract may be destroyed.

1.8.1. Modification of the data of a smart contract. The persistent data of a smart contract is usually modified as a result of executing the code of the smart contract in TVM while processing a transaction, triggered by an inbound message to the smart contract. More specifically, the code of the smart contract has access to the old persistent storage of the smart contract via TVM control register `c4`, and may modify the persistent storage by storing another value into `c4` before normal termination.

Normally, there are no other ways to modify the data of an existing smart contract. If the code of the smart contract does not provide any ways to modify the persistent data (e.g., if it is a simple wallet smart contract as described in **1.7.6**, which initializes the persistent data with the user's public key and does not intend to ever change it), then it will be effectively immutable—unless the code of the smart contract is modified first.

1.8.2. Modification of the code of a smart contract. Similarly, the code of an existing smart contract may be modified only if some provisions for such an upgrade are present in the current code. The code is modified by invoking TVM primitive `SETCODE`, which sets the root of the code for the current smart contract from the top value in the TVM stack. The modification is applied only after the normal termination of the current transaction.

Typically, if the developer of a smart contract wants to be able to upgrade its code in the future, she provides a special "code upgrade method" in the original code of the smart contract, which invokes `SETCODE` in response to certain inbound "code upgrade" messages, using the new code sent in the message itself as an argument to `SETCODE`. Some provisions must be made to protect the smart contract from unauthorized replacement of the code; otherwise, control of the smart contract and the funds on its balance could be lost. For example, code upgrade messages might be accepted only from a trusted source address, or they might be protected by requiring a valid cryptographic signature and a correct sequence number.

1.8.3. Keeping the code or data of the smart contract outside the blockchain. The code or data of the smart contract may be kept outside the blockchain and be represented only by their hashes. In such cases, only empty inbound messages may be processed, as well as messages carrying a correct copy of the smart-contract code (or its portion relevant for processing the specific message) and its data inside special fields. An example of such a situation is given by the uninitialized smart contracts and constructor messages described in **1.7**.

1.8.4. Using code libraries. Some smart contracts may share the same code, but use different data. One example of this is wallet smart contracts (cf. **1.7.6**), which are likely to use the same code (throughout all wallets created by the same software), but with different data (because each wallet must use its own pair of cryptographic keys). In this case, the code for all the wallet smart contracts is best committed by the developer into a shared

library; this library would reside in the masterchain, and be referred to by its hash using a special "external library cell reference" as the root of the code of each wallet smart contract (or as a subtree inside that code).

Notice that even if the library code becomes unavalable—for example, because its developer stops paying for its storage in the masterchain—it is still possible to use the smart contracts referring to this library, either by committing the library again into the masterchain, or by including its relevant parts inside a message sent to the smart contract. This external cell reference resolution mechanism is discussed in more detail later in **4.4.3**.

1.8.5. Destroying smart contracts. Notice that a smart contract cannot really be destroyed until its balance becomes zero or negative. It may become negative as a result of collecting persistent storage payments, or after sending a value-bearing outbound message transferring almost all of its previous balance.

For example, a user may decide to transfer all remaining funds from her wallet to another wallet or smart contract. This may be useful, for instance, if one wants to upgrade the wallet, but the wallet smart contract does not have any provisions for future upgrades; then one can simply create a new wallet and transfer all funds to it.

1.8.6. Frozen accounts. When the balance of an account becomes non-positive after a transaction, or smaller than a certain workchain-dependent minimum, the account is *frozen* by replacing all its code and data by a single 32-byte hash. This hash is kept afterwards for some time (e.g., a couple of months) to prevent recreation of the smart contract by its original creating transaction (which still has the correct hash, equal to the account address), and to allow its owner to recreate the account by transferring some funds and sending a message containing the account's code and data, to be reinstated in the blockchain. In this respect, frozen accounts are similar to uninitialized accounts; however, the hash of the correct code and data for a frozen account is not necessarily equal to the account address, but is kept separately.

Notice that frozen accounts may have a negative balance, indicating that persistent storage payments are due. An account cannot be unfrozen until its balance becomes positive and larger than a prescribed minimum value.

2 Message forwarding and delivery guarantees

This chapter discusses the forwarding of messages inside the TON Blockchain, including the Hypercube Routing (HR) and Instant Hypercube Routing (IHR) protocols. It also describes the provisions required to implement the message delivery guarantees and the FIFO ordering guarantee.

2.1 Message addresses and next-hop computation

This section explains the computation of transit and next-hop addresses by the variant of the hypercube routing algorithm employed in TON Blockchain. The hypercube routing protocol itself, which uses the concepts and next-hop address computation algorithm introduced in this section, is presented in the next section.

2.1.1. Account addresses. The *source address* and *destination address* are always present in any message. Normally, they are *(full) account addresses*. A full account address consists of a *workchain_id* (a signed 32-bit big-endian integer defining a workchain), followed by a (usually) 256-bit *internal address* or *account identifier account_id* (which may also be interpreted as an unsigned big-endian integer) defining the account within the chosen workchain.

Different workchains may use account identifiers that are shorter or longer than the "standard" 256 bits used in the masterchain ($workchain\_id = -1$) and in the basic workchain ($workchain\_id = 0$). To this end, the masterchain state contains a list of all workchains defined so far, along with their account identifier lengths. An important restriction is that the *account_id* for any workchain must be at least 64 bits long.

In what follows, we often consider only the case of 256-bit account addresses for simplicity. Only the first 64 bits of the *account_id* are relevant for the purposes of message routing and shardchain splitting.

2.1.2. Source and destination addresses of a message. Any message has both a *source address* and a *destination address*. Its source address is the address of the account (smart contract) that has created the message while processing some transaction; the source address cannot be changed or set arbitrarily, and smart contracts heavily rely on this property. By

contrast, when a message is created, any well-formed destination address may be chosen; after that, the destination address cannot be changed.

2.1.3. External messages with no source or destination address. Some messages can have no source or no destination address (though at least one of them must be present), as indicated by special flags in the message header. Such messages are the *external messages* intended for the interaction of the TON Blockchain with the outside world—human users and their cryptowallet applications, off-chain and mixed applications and services, other blockchains, and so on.

External messages are never routed inside the TON Blockchain. Instead, "messages from nowhere" (i.e., with no source address) are directly included into the *InMsgDescr* of a destination shardchain block (provided some conditions are met) and processed by a transaction in that very block. Similarly, "messages to nowhere" (i.e., with no TON Blockchain destination address), also known as *log messages*, are also present only in the block containing the transaction that generated such a message.[12]

Therefore, external messages are almost irrelevant for the discussion of message routing and message delivery guarantees. In fact, the message delivery guarantees for outbound external messages are trivial (at most, the message must be included into the *LogMsg* part of the block), and for inbound external messages there are none, since the validators of a shardchain block are free to include or ignore suggested inbound external messages at their discretion (e.g., according to the processing fee offered by the message).[13]

In what follows, we focus on "usual" or "internal" messages, which have both a source and a destination address.

2.1.4. Transit and next-hop addresses. When a message needs to be routed through intermediate shardchains before reaching its intended desti-

[12] "Messages to nowhere" may have some special fields in their body indicating their destination outside the TON Blockchain—for instance, an account in some other blockchain, or an IP address and port—which may be interpreted by the third-party software appropriately. Such fields are ignored by the TON Blockchain.

[13] The problem of bypassing possible validator censorship—which could happen, for instance, if all validators conspire not to include external messages sent to accounts belonging to some set of blacklisted accounts—is dealt with separately elsewhere. The main idea is that the validators may be forced to promise to include a message with a known hash in a future block, without knowing anything about the identity of the sender or the receiver; they will have to keep this promise afterwards when the message itself with pre-agreed hash is presented.

nation, it is assigned a *transit address* and a *next-hop address* in addition to the (immutable) source and destination addresses. When a copy of the message resides inside a transit shardchain awaiting its relay to its next hop, the *transit address* is its intermediate address lying in the transit shardchain, as if belonging to a special message-relay smart contract whose only job is to relay the unchanged message to the next shardchain on the route. The *next-hop address* is the address in a neighboring shardchain (or, on some rare occasions, in the same shardchain) to which the message needs to be relayed. After the message is relayed, the next-hop address usually becomes the transit address of the copy of the message included in the next shardchain.

Immediately after an outbound message is created in a shardchain (or in the masterchain), its transit address is set to its source address.[14]

2.1.5. Computation of the next-hop address for hypercube routing. The TON Blockchain employs a variant of hypercube routing. This means that the next-hop address is computed from the transit address (originally equal to the source address) as follows:

1. The (big-endian signed) 32-bit *workchain_id* components of both the transit address and destination address are split into groups of n_1 bits (currently, $n_1 = 32$), and they are scanned from the left (i.e., the most significant bits) to the right. If one of the groups in the transit address differs from the corresponding group in the destination address, then the value of this group in the transit address is replaced by its value in the destination address to compute the next-hop address.

2. If the *workchain_id* parts of the transit and destination addresses match, then a similar process is applied to the *account_id* parts of the addresses: The *account_id* parts, or rather their first (most significant) 64 bits, are split into groups of n_2 bits (currently, $n_2 = 4$ bit groups are used, corresponding to the hexadecimal digits of the address) starting from the most significant bit, and are compared starting from the left. The first group that differs is replaced in the transit address with its value in the destination address to compute the next-hop address.

3. If the first 64 bits of the *account_id* parts of the transit and destination addresses match as well, then the destination account belongs

[14]However, the internal routing process described in **2.1.11** is applied immediately after that, which may further modify the transit address.

to the current shardchain, and the message should not be forwarded outside the current shardchain at all. Instead, it must be processed by a transaction inside it.

2.1.6. Notation for the next-hop address. We denote by

$$\textsc{NextHop}(\xi, \eta) \tag{13}$$

the next-hop address computed for current (source or transit) address ξ and destination address η.

2.1.7. Support for anycast addresses. "Large" smart contracts, which can have separate instances in different shardchains, may be reached using *anycast destination addresses*. These addresses are supported as follows.

An anycast address (η, d) consists of a usual address η along with its "splitting depth" $d \leq 31$. The idea is that the message may be delivered to any address differing from η only in the first d bits of the internal address part (i.e., not including the workchain identifier, which must match exactly). This is achieved as follows:

- The effective destination address $\tilde{\eta}$ is computed from (η, d) by replacing the first d bits of the internal address part of η with the corresponding bits taken from the source address ξ.
- All computations of $\textsc{NextHop}(\nu, \eta)$ are replaced by $\textsc{NextHop}(\nu, \tilde{\eta})$, for $\nu = \xi$ as well as for all other intermediate addresses ν. In this way, Hypercube Routing or Instant Hypercube Routing will ultimately deliver the message to the shardchain containing $\tilde{\eta}$.
- When the message is processed in its destination shardchain (the one containing address $\tilde{\eta}$), it may be processed by an account η' of the same shardchain differing from η and $\tilde{\eta}$ only in the first d bits of the internal address part. More precisely, if the common shard address prefix is s, so that only internal addresses starting with binary string s belong to the destination shard, then η' is computed from η by replacing the first $\min(d, |s|)$ bits of the internal address part of η with the corresponding bits of s.

That said, we tacitly ignore the existence of anycast addresses and the additional processing they require in the following discussions.

2.1.8. Hamming optimality of the next-hop address algorithm. Notice that the specific hypercube routing next-hop computation algorithm explained in **2.1.5** may potentially be replaced by another algorithm, provided it satisfies certain properties. One of these properties is the *Hamming optimality*, meaning that the Hamming (L_1) distance from ξ to η equals the sum of Hamming distances from ξ to $\textsc{NextHop}(\xi,\eta)$ and from $\textsc{NextHop}(\xi,\eta)$ to η:

$$\|\xi-\eta\|_1 = \big\|\xi - \textsc{NextHop}(\xi,\eta)\big\|_1 + \big\|\textsc{NextHop}(\xi,\eta)-\eta\big\|_1 \tag{14}$$

Here $\|\xi-\eta\|_1$ is the *Hamming distance* between ξ and η, equal to the number of bit positions in which ξ and η differ:[15]

$$\|\xi-\eta\|_1 = \sum_i |\xi_i-\eta_i| \tag{15}$$

Notice that in general one should expect only an inequality in (14), following from the triangle inequality for the L_1-metric. Hamming optimality essentially means that $\textsc{NextHop}(\xi,\eta)$ lies on one of the (Hamming) shortest paths from ξ to η. It can also be expressed by saying that $\nu = \textsc{NextHop}(\xi,\eta)$ is always obtained from ξ by changing the values of bits at some positions to their values in η: for any bit position i, we have $\nu_i=\xi_i$ or $\nu_i=\eta_i$.[16]

2.1.9. Non-stopping of NextHop. Another important property of the NextHop is its *non-stopping*, meaning that $\textsc{NextHop}(\xi,\eta)=\xi$ is possible only when $\xi=\eta$. In other words, if we have not yet arrived at η, the next hop cannot coincide with our current position.

This property implies that the path from ξ to η—i.e., the sequence of intermediate addresses $\xi^{(0)}:=\xi$, $\xi^{(n)}:=\textsc{NextHop}(\xi^{(n-1)},\eta)$—will gradually stabilize at η: for some $N\geq 0$, we have $\xi^{(n)}=\eta$ for all $n\geq N$. Indeed, one can always take $N:=\|\xi-\eta\|_1$.

[15] When the addresses involved are of different lengths (e.g., because they belong to different workchains), one should consider only the first 96 bits of the addresses in the above formula.

[16] Instead of Hamming optimality, we might have considered the equivalent property of *Kademlia optimality*, written for the Kademlia (or weighted L_1) distance as given by $\|\xi-\eta\|_K := \sum_i 2^{-i}|\xi_i-\eta_i|$ instead of the Hamming distance.

2.1.10. Convexity of the HR path with respect to sharding. A consequence of Hamming optimality property (14) is what we call the *convexity* of the path from ξ to η with respect to sharding. Namely, if $\xi^{(0)} := \xi$, $\xi^{(n)} := \textsc{NextHop}(\xi^{(n-1)}, \eta)$ is the computed path from ξ to η, and N is the first index such that $\xi^{(N)} = \eta$, and S is a shard of some workchain in any shard configuration, then the indices i with $\xi^{(i)}$ residing in shard S constitute a subinterval in $[0, N]$. In other words, if integers $0 \le i \le j \le k \le N$ are such that $\xi^{(i)}, \xi^{(k)} \in S$, then $\xi^{(j)} \in S$ as well.

This convexity property is important for some proofs related to message forwarding in the presence of dynamic sharding.

2.1.11. Internal routing. Notice that the next-hop address computed according to the rules defined in **2.1.5** may belong to the same shardchain as the current one (i.e., the one containing the transit address). In that case, the "internal routing" occurs immediately, the transit address is replaced by the value of the computed next-hop address, and the next-hop address computation step is repeated until a next-hop address lying outside the current shardchain is obtained. The message is then kept in the transit output queue according to its computed next-hop address, with its last computed transit address as the "intermediate owner" of the transit message. If the current shardchain splits into two shardchains before the message is forwarded further, it is the shardchain containing the intermediate owner that inherits this transit message.

Alternatively, we might go on computing the next-hop addresses only to find out that the destination address already belongs to the current shardchain. In that case, the message will be processed (by a transaction) inside this shardchain instead of being forwarded further.

2.1.12. Neighboring shardchains. Two shards in a shard configuration—or the two corresponding shardchains—are said to be *neighbors*, or *neighboring shardchains*, if one of them contains a next-hop address for at least one combination of allowed source and destination addresses, while the other contains the transit address for the same combination. In other words, two shardchains are neighbors if a message can be forwarded directly from one of them into the other via Hypercube Routing.

The masterchain is also included in this definition, as if it were the only shardchain of the workchain with *workchain_id* $= -1$. In this respect, it is a neighbor of all the other shardchains.

2.1.13. Any shard is a neighbor of itself. Notice that a shardchain is always considered a neighbor of itself. This may seem redundant, because we always repeat the next-hop computation described in **2.1.5** until we obtain a next-hop address outside the current shardchain (cf. **2.1.11**). However, there are at least two reasons for such an arrangement:

- Some messages have the source and the destination address inside the same shardchain, at least when the message is created. However, if such a message is not processed immediately in the same block where it has been created, it must be added to the outbound message queue of its shardchain, and be imported as an inbound message (with an entry in the *InMsgDescr*) in one of the subsequent blocks of the same shardchain.[17]

- Alternatively, the next-hop address may originally be in some other shardchain that later gets merged with the current shardchain, so that the next hop becomes inside the same shardchain. Then the message will have to be imported from the outbound message queue of the merged shardchain, and forwarded or processed accordingly to its next-hop address, even though they reside now inside the same shardchain.

2.1.14. Hypercube Routing and the ISP. Ultimately, the Infinite Sharding Paradigm (ISP) applies here: a shardchain should be considered a provisional union of accountchains, grouped together solely to minimize the block generation and transmission overhead.

The forwarding of a message runs through several intermediate accountchains, some of which can happen to lie in the same shard. In this case, once a message reaches an accountchain lying in this shard, it is immediately ("internally") routed inside that shard until the last accountchain lying in the same shard is reached (cf. **2.1.11**). Then the message is enqueued in the output queue of that last accountchain.[18]

2.1.15. Representation of transit and next-hop addresses. Notice that the transit and next-hop addresses differ from the source address only

[17]Notice that the next-hop and internal-routing computations are still applied to such messages, since the current shardchain may be split before the message is processed. In this case, the new sub-shardchain containing the destination address will inherit the message.

[18]We may define the (virtual) output queue of an account(chain) as the subset of the *OutMsgQueue* of the shard currently containing that account that consists of messages with transit addresses equal to the address of the account.

in the *workchain_id* and in the first (most significant) 64 bits of the account address. Therefore, they may be represented by 96-bit strings. Furthermore, their *workchain_id* usually coincides with the *workchain_id* of either the source address or the destination address; a couple of bits may be used to indicate this situation, thus further reducing the space required to represent the transit and next-hop addresses.

In fact, the required storage may be reduced even further by observing that the specific hypercube routing algorithm described in **2.1.5** always generates intermediate (i.e., transit and next-hop) addresses that coincide with the destination address in their first k bits, and with the source address in their remaining bits. Therefore, one might use just the values $0 \leq k_{\mathrm{tr}}, k_{\mathrm{nh}} \leq 96$ to fully specify the transit and next-hop addresses. One might also notice that $k' := k_{\mathrm{nh}}$ turns out to be a fixed function of $k := k_{\mathrm{tr}}$ (for instance, $k' = k + n_2 = k + 4$ for $k \geq 32$), and therefore include only one 7-bit value of k in the serialization.

Such optimizations have the obvious disadvantage that they rely too much on the specific routing algorithm used, which can be changed in the future, so they are used in **3.1.15** with a provision to specify more general intermediate addresses if necessary.

2.1.16. Message envelopes. The transit and next-hop addresses of a forwarded message are not included in the message itself, but are kept in a special *message envelope*, which is a cell (or a cell slice) containing the transit and next-hop addresses with the above optimizations, some other information relevant for forwarding and processing, and a reference to a cell containing the unmodified original message. In this way, a message can easily be "extracted" from its original envelope (e.g., the one present in the *InMsgDescr*) and be put into another envelope (e.g., before being included into the *OutMsgQueue*).

In the representation of a block as a tree, or rather a DAG, of cells, the two different envelopes will contain references to a shared cell with the original message. If the message is large, this arrangement avoids the need to keep more than one copy of the message in the block.

2.2 Hypercube Routing protocol

This section exposes the details of the hypercube routing protocol employed by the TON Blockchain to achieve guaranteed delivery of messages between

smart contracts residing in arbitrary shardchains. For the purposes of this document, we will refer to the variant of hypercube routing employed by the TON Blockchain as Hypercube Routing (HR).

2.2.1. Message uniqueness. Before continuing, let us observe that any (internal) message is *unique*. Recall that a message contains its full source address along with its logical creation time, and all outbound messages created by the same smart contract have strictly increasing logical creation times (cf. **1.4.6**); therefore, the combination of the full source address and the logical creation time uniquely defines the message. Since we assume the chosen hash function SHA256 to be collision resistant, *a message is uniquely determined by its hash*, so we can identify two messages if we know that their hashes coincide.

This does not extend to external messages "from nowhere", which have no source addresses. Special care must be taken to prevent replay attacks related to such messages, especially by designers of user wallet smart contracts. One possible solution is to include a sequence number in the body of such messages, and keep the count of external messages already processed inside the smart-contract persistent data, refusing to process an external message if its sequence number differs from this count.

2.2.2. Identifying messages with equal hashes. The TON Blockchain assumes that two messages with the same hashes coincide, and treats either of them as a redundant copy of the other. As explained above in **2.2.1**, this does not lead to any unexpected effects for internal messages. However, if one sends two coinciding "messages from nowhere" to a smart contract, it may happen that only one of them will be delivered—or both. If their action is not supposed to be idempotent (i.e., if processing the message twice has a different effect from processing it once), some provisions should be made to distinguish the two messages, for instance by including a sequence number in them.

In particular, the *InMsgDescr* and *OutMsgDescr* use the (unenveloped) message hash as a key, tacitly assuming that distinct messages have distinct hashes. In this way, one can trace the path and the fate of a message across different shardchains by looking up the message hash in the *InMsgDescr* and *OutMsgDescr* of different blocks.

2.2.3. The structure of *OutMsgQueue*. Recall that the outbound messages — both those created inside the shardchain, and transit messages pre-

viously imported from a neighboring shardchain to be relayed to the next-hop shardchain — are accumulated in the *OutMsgQueue*, which is part of the *state* of the shardchain (cf. **1.2.7**). In contrast with *InMsgDescr* and *OutMsgDescr*, the key in *OutMsgQueue* is not the message hash, but its next-hop address—or at least its first 96 bits—concatenated with the message hash.

Furthermore, the *OutMsgQueue* is not just a dictionary (hashmap), mapping its keys into (enveloped) messages. Rather, it is a *min-augmented dictionary with respect to the logical creation time*, meaning that each node of the Patricia tree representing *OutMsgQueue* has an additional value (in this case, an unsigned 64-bit integer), and that this augmentation value in each fork node is set to be equal to the minimum of the augmentation values of its children. The augmentation value of a leaf equals the logical creation time of the message contained in that leaf; it need not be stored explicitly.

2.2.4. Inspecting the *OutMsgQueue* of a neighbor. Such a structure for the *OutMsgQueue* enables the validators of a neighboring shardchain to inspect it to find its part (Patricia subtree) relevant to them (i.e., consisting of messages with the next-hop address belonging to the neighboring shard in question—or having the next-hop address with a given binary prefix), as well as quickly compute the "oldest" (i.e., with the minimum logical creation time) message in that part.

Furthermore, the shard validators do not even need to track the total state of all their neighboring shardchains—they only need to keep and update a copy of their *OutMsgQueue*, or even of its subtree related to them.

2.2.5. Logical time monotonicity: importing the oldest message from the neighbors. The first fundamental local condition of message forwarding, called *(message import) (logical time) monotonicity condition*, may be summarized as follows:

> While importing messages into the *InMsgDescr* of a shardchain block from the *OutMsgQueue*s of its neighboring shardchains, the validators must import the messages in the increasing order of their logical time; in the case of a tie, the message with the smaller hash is imported first.

More precisely, each shardchain block contains the hash of a masterchain block (assumed to be "the latest" masterchain block at the time of the shardchain block's creation), which in turn contains the hashes of the most recent

shardchain blocks. In this way, each shardchain block indirectly "knows" the most recent state of all other shardchains, and especially its neighboring shardchains, including their *OutMsgQueue*s.[19]

Now an alternative equivalent formulation of the monotonicity condition is as follows:

> If a message is imported into the *InMsgDescr* of the new block, its logical creation time cannot be greater than that of any message left unimported in the *OutMsgQueue* of the most recent state of any of the neighboring shardchains.

It is this form of the monotonicity condition that appears in the local consistency conditions of the TON Blockchain blocks and is enforced by the validators.

2.2.6. Witnesses to violations of the message import logical time monotonicity condition. Notice that if this condition is not fulfilled, a small Merkle proof witnessing its failure may be constructed. Such a proof will contain:

- A path in the *OutMsgQueue* of a neighbor from the root to a certain message m with small logical creation time.
- A path in the *InMsgDescr* of the block under consideration showing that the key equal to $\textsc{Hash}(m)$ is absent in *InMsgDescr* (i.e., that m has not been included in the current block).
- A proof that m has not been included in a preceding block of the same shardchain, using the block header information containing the smallest and the largest logical time of all messages imported into the block (cf. **2.3.4**–**2.3.7** for more information).
- A path in *InMsgDescr* to another included message m', such that either $\textsc{Lt}(m') > \textsc{Lt}(m)$, or $\textsc{Lt}(m') = \textsc{Lt}(m)$ and $\textsc{Hash}(m') > \textsc{Hash}(m)$.

2.2.7. Deleting a message from *OutMsgQueue*. A message must be deleted from *OutMsgQueue* sooner or later; otherwise, the storage used by

[19] In particular, if the hash of a recent block of a neighboring shardchain is not yet reflected in the latest masterchain block, its modifications to *OutMsgQueue* must not be taken into account.

OutMsgQueue would grow to infinity. To this end, several "garbage collection rules" are introduced. They allow the deletion of a message from *OutMsgQueue* during the evaluation of a block only if an explicit special "delivery record" is present in the *OutMsgDescr* of that block. This record contains either a reference to the neighboring shardchain block that has included the message into its *InMsgDescr* (the hash of the block is sufficient, but collated material for the block may contain the relevant Merkle proof), or a Merkle proof of the fact that the message has been delivered to its final destination via Instant Hypercube Routing.

2.2.8. Guaranteed message delivery via Hypercube Routing. In this way, a message cannot be deleted from the outbound message queue unless it has been either relayed to its next-hop shardchain or delivered to its final destination (cf. **2.2.7**). Meanwhile, the message import monotonicity condition (cf. **2.2.5**) ensures that any message will sooner or later be relayed into the next shardchain, taking into account other conditions which require the validators to use at least half of the block's space or gas limits for importing inbound internal messages (otherwise the validators might choose to create empty blocks or import only external messages even in the presence of non-empty outbound message queues at their neighbors).

2.2.9. Message processing order. When several imported messages are processed by transactions inside a block, the *message processing order conditions* ensure that older messages are processed first. More precisely, if a block contains two transactions t and t' of the same account, which process inbound messages m and m', respectively, and $\text{Lt}(m) < \text{Lt}(m')$, then we must have $\text{Lt}(t) < \text{Lt}(t')$.

2.2.10. FIFO guarantees of Hypercube Routing. The message processing order conditions (cf. **2.2.9**), along with the message import monotonicity conditions (cf. **2.2.5**), imply the *FIFO guarantees for Hypercube Routing*. Namely, if a smart contract ξ creates two messages m and m' with the same destination η, and m' is generated later than m (meaning that $m \prec m'$, hence $\text{Lt}(m) < \text{Lt}(m')$), then m will be processed by η before m'. This is so because both messages will follow the same routing steps on the path from ξ to η (the Hypercube Routing algorithm described in **2.1.5** is deterministic), and in all outbound queues and inbound message descriptions m' will appear "after" m.[20]

[20]This statement is not as trivial as it seems at first, because some of the shardchains

If message m' can be delivered to B via Instant Hypercube Routing, this is not necessarily true anymore. Therefore, a simple way of ensuring FIFO message delivery discipline between a pair of smart contracts consists in setting a special bit in the message header preventing its delivery via IHR.

2.2.11. Delivery uniqueness guarantees of Hypercube Routing. Notice that the message import monotonicity conditions also imply the *uniqueness* of the delivery of any message via Hypercube Routing—i.e., that it cannot be imported and processed by the destination smart contract more than once. We will see later in **2.3** that enforcing delivery uniqueness when both Hypercube Routing and Instant Hypercube Routing are active is more complicated.

2.2.12. An overview of Hypercube Routing. Let us summarize all routing steps performed to deliver an internal message m created by source account ξ_0 to destination account η. We denote by $\xi_{k+1} := \textsc{NextHop}(\xi_k, \eta)$, $k = 0, 1, 2, \ldots$ the intermediate addresses dictated by HR for forwarding the message m to its final destination η. Let S_k be the shard containing ξ_k.

- [Birth] — Message m with destination η is created by a transaction t belonging to an account ξ_0 residing in some shardchain S_0. The logical creation time $\textsc{Lt}(m)$ is fixed at this point and included into the message m.

- [ImmediateProcessing?] — If the destination η resides in the same shardchain S_0, the message may be processed in the same block it was generated in. In this case, m is included into *OutMsgDescr* with a flag indicating it has been processed in this very block and need not be forwarded further. Another copy of m is included into *InMsgDescr*, along with the usual data describing the processing of inbound messages. (Notice that m is not included into the *OutMsgQueue* of S_0.)

involved may split or merge during the routing. A correct proof may be obtained by adopting the ISP perspective to HR as explained in **2.1.14** and observing that m' will always be behind m, either in terms of the intermediate accountchain reached or, if they happen to be in the same accountchain, in terms of logical creation time.

A crucial observation is that "at any given moment of time" (logically; a more precise description would be "in the total state obtained after processing any causally closed subset $\mathscr{F}$ of blocks"), the intermediate accountchains belonging to the same shard are contiguous on the path from ξ to η (i.e., cannot have accountchains belonging to some other shard in between). This is a "convexity property" (cf. **2.1.10**) of the Hypercube Routing algorithm described in **2.1.5**.

- [InitialInternalRouting] — If m either has a destination outside S_0, or is not processed in the same block where it was generated, the internal routing procedure described in **2.1.11** is applied, until an index k is found such that ξ_k lies in S_0, but $\xi_{k+1} = \text{NextHop}(\xi_k, \eta)$ does not (i.e., $S_k = S_0$, but $S_{k+1} \neq S_0$). Alternatively, this process stops if $\xi_k = \eta$ or ξ_k coincides with η in its first 96 bits.

- [OutboundQueuing] — The message m is included into *OutMsgDescr* (with the key equal to its hash), with an envelope containing its transit address ξ_k and next-hop address ξ_{k+1} as explained in **2.1.16** and **2.1.15**. The same enveloped message is also included in the *OutMsgQueue* of the state of S_k, with the key equal to the concatenation of the first 96 bits of its next-hop address ξ_{k+1} (which may be equal to η if η belongs to S_k) and the message hash $\text{Hash}(m)$.

- [QueueWait] — Message m waits in the *OutMsgQueue* of shardchain S_k to be forwarded further. In the meantime, shardchain S_k may split or merge with other shardchains; in that case, the new shard S'_k containing the transit address ξ_k inherits m in its *OutMsgQueue*.

- [ImportInbound] — At some point in the future, the validators for the shardchain S_{k+1} containing the next-hop address ξ_{k+1} scan the *OutMsgQueue* in the state of shardchain S_k and decide to import message m in keeping with the monotonicity condition (cf. **2.2.5**) and other conditions. A new block for shardchain S_{k+1} is generated, with an enveloped copy of m included in its *InMsgDescr*. The entry in *InMsgDescr* contains also the *reason* for importing m into this block, with a hash of the most recent block of shardchain S'_k, and the previous next-hop and transit addresses ξ_k and ξ_{k+1}, so that the corresponding entry in the *OutMsgQueue* of S'_k can be easily located.

- [Confirmation] — This entry in the *InMsgDescr* of S_{k+1} also serves as a confirmation for S'_k. In a later block of S'_k, message m must be removed from the *OutMsgQueue* of S'_k; this modification is reflected in a special entry in the *OutMsgDescr* of the block of S'_k that performs this state modification.

- [Forwarding?] — If the final destination η of m does not reside in S_{k+1}, the message is *forwarded*. Hypercube Routing is applied until

some ξ_l, $l > k$, and $\xi_{l+1} = \text{NextHop}(\xi_l, \eta)$ are obtained, such that ξ_l lies in S_{k+1}, but ξ_{l+1} does not (cf. **2.1.11**). After that, a newly-enveloped copy of m with transit address set to ξ_l and next-hop address ξ_{l+1} is included into both the *OutMsgDescr* of the current block of S_{k+1} and the *OutMsgQueue* of the new state of S_{k+1}. The entry of m in *InMsgDescr* contains a flag indicating that the message has been forwarded; the entry in *OutMsgDescr* contains the newly-enveloped message and a flag indicating that this is a forwarded message. Then all the steps starting from [OutboundQueueing] are repeated, for l instead of k.

- [Processing?] — If the final destination η of m resides in S_{k+1}, then the block of S_{k+1} that imported the message must process it by a transaction t included in the same block. In this case, *InMsgDescr* contains a reference to t by its logical time $\text{Lt}(t)$, and a flag indicating that the message has been processed.

The above message routing algorithm does not take into account some further modifications required to implement Instant Hypercube Routing (IHR). For instance, a message may be *discarded* after being imported (listed in *InMsgDescr*) into its final or intermediate shardchain block, because a proof of delivery via IHR to the final destination is presented. In this case, such a proof must be included into *InMsgDescr* to explain why the message was not forwarded further or processed.

2.3 Instant Hypercube Routing and combined delivery guarantees

This section describes the Instant Hypercube Routing protocol, normally applied by TON Blockchain in parallel to the previously discussed Hypercube Routing protocol to achieve faster message delivery. However, when both Hypercube Routing and Instant Hypercube Routing are applied to the same message in parallel, achieving delivery and unique delivery guarantees is more complicated. This topic is also discussed in this section.

2.3.1. An overview of Instant Hypercube Routing. Let us explain the major steps applied when the Instant Hypercube Routing (IHR) mechanism is applied to a message. (Notice that normally both the usual HR and IHR

work in parallel for the same message; some provisions must be taken to guarantee the uniqueness of delivery of any message.)

Consider the routing and delivery of the same message m with source ξ and destination η as discussed in **2.2.12**:

- [NetworkSend] — After the validators of S_0 have agreed on and signed the block containing the creating transaction t for m, and observed that the destination η of m does not reside inside S_0, they may send a datagram (encrypted network message), containing the message m along with a Merkle proof of its inclusion into the *OutMsgDescr* of the block just generated, to the validator group of the shardchain T currently owning the destination η.

- [NetworkReceive] — If the validators of shardchain T receive such a message, they check its validity starting from the most recent masterchain block and the shardchain block hashes listed in it, including the most recent "canonical" block of shardchain S_0 as well. If the message is invalid, they silently discard it. If that block of shardchain S_0 has a larger sequence number than the one listed in the most recent masterchain block, they may either discard it or postpone the verification until the next masterchain block appears.

- [InclusionConditions] — The validators check inclusion conditions for message m. In particular, they must check that this message has not been delivered before, and that the *OutMsgQueue*s of the neighbors do not have unprocessed outbound messages with destinations in T with smaller logical creation times than $\text{Lt}(m)$.

- [Deliver] — The validators deliver and process the message, by including it into the *InMsgDescr* of the current shardchain block along with a bit indicating that it is an IHR message, the Merkle proof of its inclusion into the *OutMsgDescr* of the original block, and the logical time of the transaction t' processing this inbound message into the currently generated block.

- [Confirm] — Finally, the validators send encrypted datagrams to all the validator groups of the intermediate shardchains on the path from ξ to η, containing a Merkle proof of the inclusion of message m into the *InMsgDescr* of its final destination. The validators of an intermediate shardchain may use this proof to *discard* the copy of message

m travelling by the rules of HR, by importing the message into their *InMsgDescr* along with the Merkle proof of final delivery and setting a flag indicating that the message has been discarded.

The overall procedure is even simpler than that for Hypercube Routing. Notice, however, that IHR comes with no delivery or FIFO guarantees: the network datagram may be lost in transit, or the validators of the destination shardchain may decide not to act on it, or they may discard it due to buffer overflow. This is the reason why IHR is used as a complement to HR, and not as a replacement.

2.3.2. Overall eventual delivery guarantees. Notice that the combination of HR and IHR guarantees the ultimate delivery of any internal message to its final destination. Indeed, the HR by itself is guaranteed to deliver any message eventually, and the HR for message m can be cancelled at an intermediate stage only by a Merkle proof of delivery of m to its final destination (via IHR).

2.3.3. Overall unique delivery guarantees. However, the *uniqueness* of message delivery for the combination of HR and IHR is more difficult to achieve. In particular, one must check the following conditions, and, if necessary, be able to provide short Merkle proofs that they do or don't hold:

- When a message m is imported into its next intermediate shardchain block via HR, we must check that m has not already been imported via HR.
- When m is imported and processed in its final destination shardchain, we must check that m has not already been processed. If it has, there are three subcases:
  - If m is being considered for import via HR, and it has already been imported via HR, it must not be imported at all.
  - If m is being considered for import via HR, and it has already been imported via IHR (but not HR), then it must be imported and immediately discarded (without being processed by a transaction). This is necessary to remove m from the *OutMsgQueue* of its previous intermediate shardchain.

– If m is being considered for import via IHR, and it has already been imported via either IHR or HR, it must not be imported at all.

2.3.4. Checking whether a message has already been delivered to its final destination. Consider the following general algorithm for checking whether a message m has already been delivered to its final destination η: One can simply scan the last several blocks belonging to the shardchain containing the destination address, starting from the latest block and working backwards through the previous block references. (If there are two previous blocks—i.e., if a shardchain merge event occurred at some point—one would follow the chain containing the destination address.) The *InMsgDescr* of each of these blocks can be checked for an entry with key $\textsc{Hash}(m)$. If such an entry is found, the message m has already been delivered, and we can easily construct a Merkle proof of this fact. If we do not find such an entry before arriving at a block B with $\textsc{Lt}^+(B) < \textsc{Lt}(m)$, implying that m could not be delivered in B or any of its predecessors, then the message m definitely has not been delivered yet.

The obvious disadvantage of this algorithm is that, if message m is very old (and most likely delivered a long time ago), meaning that it has a small value of $\textsc{Lt}(m)$, then a large number of blocks will need to be scanned before yielding an answer. Furthermore, if the answer is negative, the size of the Merkle proof of this fact will increase linearly with the number of blocks scanned.

2.3.5. Checking whether an IHR message has already been delivered to its final destination. To check whether an IHR message m has already been delivered to its destination shardchain, we can apply the general algorithm described above (cf. **2.3.4**), modified to inspect only the last c blocks for some small constant c (say, $c = 8$). If no conclusion can be reached after inspecting these blocks, then the validators for the destination shardchain may simply discard the IHR message instead of spending more resources on this check.

2.3.6. Checking whether an HR message has already been delivered via HR to its final destination or an intermediate shardchain. To check whether an HR-received message m (or rather, a message m being considered for import via HR) has already been imported via HR, we can use the following algorithm: Let ξ_k be the transit address of m (belonging to

a neighboring shardchain S_k) and ξ_{k+1} be its next-hop address (belonging to the shardchain under consideration). Since we are considering the inclusion of m, m must be present in the *OutMsgQueue* of the most recent state of shardchain S_k, with ξ_k and ξ_{k+1} indicated in its envelope. In particular, (a) the message has been included into *OutMsgQueue*, and we may even know when, because the entry in *OutMsgQueue* sometimes contains the logical time of the block where it has been added, and (b) it has not yet been removed from *OutMsgQueue*.

Now, the validators of the neighboring shardchain are required to remove a message from *OutMsgQueue* as soon as they observe that message (with transit and next-hop addresses ξ_k and ξ_{k+1} in its envelope) has been imported into the *InMsgDescr* of the message's next-hop shardchain. Therefore, (b) implies that the message could have been imported into the *InMsgDescr* of a preceding block only if this preceding block is very new (i.e., not yet known to the most recent neighboring shardchain block). Therefore, only a very limited number of preceding blocks (typically one or two, at most) need to be scanned by the algorithm described in **2.3.4** to conclude that the message has not yet been imported.[21] In fact, if this check is performed by the validators or collators for the current shardchain themselves, it can be optimized by keeping in memory the *InMsgDescr*s of the several latest blocks.

2.3.7. Checking whether an HR message has already been delivered via IHR to its final destination. Finally, to check whether an HR message has already been delivered to its final destination via IHR, one can use the general algorithm described in **2.3.4**. In contrast with **2.3.5**, we cannot abort the verification process after scanning a fixed number of the latest blocks in the destination shardchain, because HR messages cannot be dropped without a reason.

Instead, we indirectly bound the number of blocks to be inspected by forbidding the inclusion of IHR message m into a block B of its destination shardchain if there are already more than, say, $c = 8$ blocks B' in the destination shardchain with $\text{LT}^+(B') \geq \text{LT}(m)$.

Such a condition effectively restricts the time interval after the creation of message m in which it could have been delivered via IHR, so that only a small number of blocks of the destination shardchain (at most c) will need

[21] One must not only look up the key $\text{HASH}(m)$ in the *InMsgDescr* of these blocks, but also check the intermediate addresses in the envelope of the corresponding entry, if found.

to be inspected.

Notice that this condition nicely aligns with the modified algorithm described in **2.3.5**, effectively forbidding the validators from importing the newly-received IHR message if more than $c = 8$ steps are needed to check that it had not been imported already.

3 Messages, message descriptors, and queues

This chapter presents the internal layout of individual messages, message descriptors (such as *InMsgDescr* or *OutMsgDescr*), and message queues (such as *OutMsgQueue*). Enveloped messages (cf. **2.1.16**) are also discussed here.

Notice that most general conventions related to messages must be obeyed by all shardchains, even if they do not belong to the basic shardchain; otherwise, messaging and interaction between different workchains would not be possible. It is the *interpretation* of the message contents and the *processing* of messages, usually by some transactions, that differs between workchains.

3.1 Address, currency, and message layout

This chapter begins with some general definitions, followed by the precise layout of addresses used for serializing source and destination addresses in a message.

3.1.1. Some standard definitions. For the reader's convenience, we reproduce here several general TL-B definitions.[22] These definitions are used below in the discussion of address and message layout, but otherwise are not related to the TON Blockchain.

```
unit$_ = Unit;
true$_ = True;
// EMPTY False;
bool_false$0 = Bool;
bool_true$1 = Bool;
nothing$0 {X:Type} = Maybe X;
just$1 {X:Type} value:X = Maybe X;
left$0 {X:Type} {Y:Type} value:X = Either X Y;
right$1 {X:Type} {Y:Type} value:Y = Either X Y;
pair$_ {X:Type} {Y:Type} first:X second:Y = Both X Y;

bit$_ _:(## 1) = Bit;
```

3.1.2. TL-B scheme for addresses. The serialization of source and destination addresses is defined by the following TL-B scheme:

[22] A description of an older version of TL may be found at `https://core.telegram.org/mtproto/TL`. Alternatively, an informal introduction to TL-B schemes may be found in [4, 3.3.4].

```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 9) external_address:(len * Bit)
             = MsgAddressExt;
anycast_info$_ depth:(## 5) rewrite_pfx:(depth * Bit) = Anycast;
addr_std$10 anycast:(Maybe Anycast)
   workchain_id:int8 address:uint256  = MsgAddressInt;
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9)
   workchain_id:int32 address:(addr_len * Bit) = MsgAddressInt;
_ MsgAddressInt = MsgAddress;
_ MsgAddressExt = MsgAddress;
```

The two last lines define type `MsgAddress` to be the internal union of types `MsgAddressInt` and `MsgAddressExt` (not to be confused with their external union `Either MsgAddressInt MsgAddressExt` as defined in **3.1.1**), as if the preceding four lines had been repeated with the right-hand side replaced by `MsgAddress`. In this way, type `MsgAddress` has four constructors, and types `MsgAddressInt` and `MsgAddressExt` are both subtypes of `MsgAddress`.

3.1.3. External addresses. The first two constructors, `addr_none` and `addr_extern`, are used for source addresses of "messages from nowhere" (inbound external messages), and for destination addresses of "messages to nowhere" (outbound external messages). The `addr_extern` constructor defines an "external address", which is ignored by the TON Blockchain software altogether (which treats `addr_extern` as a longer variant of `addr_none`), but may be used by external software for its own purposes. For example, a special external service may inspect the destination address of all outbound external messages found in all blocks of the TON Blockchain, and, if a special magic number is present in the `external_address` field, parse the remainder as an IP address and UDP port or a (TON Network) ADNL address, and send a datagram with a copy of the message to the network address thus obtained.

3.1.4. Internal addresses. The two remaining constructors, `addr_std` and `addr_var`, represent internal addresses. The first of them, `addr_std`, represents a signed 8-bit *workchain_id* (sufficient for the masterchain and for the basic workchain) and a 256-bit internal address in the selected workchain. The second of them, `addr_var`, represents addresses in workchains with a "large" *workchain_id*, or internal addresses of length not equal to 256. Both of these constructors have an optional `anycast` value, absent by default,

which enables "address rewriting" when present.[23]

The validators must use `addr_std` instead of `addr_var` whenever possible, but must be ready to accept `addr_var` in inbound messages. The `addr_var` constructor is intended for future extensions.

Notice that *workchain_id* must be a valid workchain identifier enabled in the current masterchain configuration, and the length of the internal address must be in the range allowed for the indicated workchain. For example, one cannot use *workchain_id* $= 0$ (basic workchain) or *workchain_id* $= -1$ (masterchain) with addresses that are not exactly 256 bits long.

3.1.5. Representing Gram currency amounts. Amounts of Grams are expressed with the aid of two types representing variable-length unsigned or signed integers, plus a type `Grams` explicitly dedicated to representing non-negative amounts of nanograms, as follows:

```
var_uint$_ {n:#} len:(#< n) value:(uint (len * 8))
         = VarUInteger n;
var_int$_ {n:#} len:(#< n) value:(int (len * 8))
        = VarInteger n;
nanograms$_ amount:(VarUInteger 16) = Grams;
```

If one wants to represent x nanograms, one selects an integer $\ell < 16$ such that $x < 2^{8\ell}$, and serializes first ℓ as an unsigned 4-bit integer, then x itself as an unsigned 8ℓ-bit integer. Notice that four zero bits represent a zero amount of Grams.

Recall (cf. [3, A]) that the original total supply of Grams is fixed at five billion (i.e., $5 \cdot 10^{18} < 2^{63}$ nanograms), and is expected to grow very slowly. Therefore, all the amounts of Grams encountered in practice will fit in unsigned or even signed 64-bit integers. The validators may use the 64-bit integer representation of Grams in their internal computations; however, the serialization of these values the blockchain is another matter.

3.1.6. Representing collections of arbitrary currencies. Recall that the TON Blockchain allows its users to define arbitrary cryptocurrencies

[23] *Address rewriting* is a feature used to implement "anycast addresses" employed by the so-called *large* or *global* smart contracts (cf. [3, 2.3.18]), which can have instances in several shardchains. When address rewriting is enabled, a message may be routed to and processed by a smart contract with an address coinciding with the destination address up to the first d bits, where $d \leq 32$ is the "splitting depth" of the smart contract indicated in the `anycast.depth` field (cf. **2.1.7**). Otherwise, the addresses must match exactly.

or tokens apart from the Gram, provided some conditions are met. Such additional cryptocurrencies are identified by 32-bit *currency_id*s. The list of defined additional cryptocurrencies is a part of the blockchain configuration, stored in the masterchain.

When some amounts of one or several such cryptocurrencies need to be represented, a dictionary (cf. [4, 3.3]) with 32-bit *currency_id*s as keys and `VarUInteger 32` values is used:

```
extra_currencies$_ dict:(HashmapE 32 (VarUInteger 32))
                 = ExtraCurrencyCollection;
currencies$_ grams:Grams other:ExtraCurrencyCollection
           = CurrencyCollection;
```

The value attached to an internal message is represented by a value of the `CurrencyCollection` type, which may describe a certain (non-negative) amount of (nano)grams as well as some additional currencies, if needed. Notice that if no additional currencies are required, `other` reduces to just one zero bit.

3.1.7. Message layout. A message consists of its *header* followed by its *body*, or *payload*. The body is essentially arbitrary, to be interpreted by the destination smart contract. The message header is standard and is organized as follows:

```
int_msg_info$0 ihr_disabled:Bool bounce:Bool
  src:MsgAddressInt dest:MsgAddressInt
  value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;
ext_in_msg_info$10 src:MsgAddressExt dest:MsgAddressInt
  import_fee:Grams = CommonMsgInfo;
ext_out_msg_info$11 src:MsgAddressInt dest:MsgAddressExt
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

```
message$_ {X:Type} info:CommonMsgInfo
  init:(Maybe (Either StateInit ^StateInit))
  body:(Either X ^X) = Message X;
```

The meaning of this scheme is as follows.

Type `Message` X describes a message with the body (or payload) of type X. Its serialization starts with `info` of type `CommonMsgInfo`, which comes in three flavors: for internal messages, inbound external messages, and outbound external messages, respectively. All of them have a source address `src` and destination address `dest`, which are external or internal according to the chosen constructor. Apart from that, an internal message may bear some `value` in Grams and other defined currencies (cf. **3.1.6**), and all messages generated inside the TON Blockchain have a logical creation time `created_lt` (cf. **1.4.6**) and creation unixtime `created_at`, both automatically set by the generating transaction. The creation unixtime equals the creation unixtime of the block containing the generating transaction.

3.1.8. Forwarding and IHR fees. Total value of an internal message. Internal messages define an `ihr_fee` in Grams, which is subtracted from the value attached to the message and awarded to the validators of the destination shardchain if they include the message by the IHR mechanism. The `fwd_fee` is the original total forwarding fee paid for using the HR mechanism; it is automatically computed from some configuration parameters and the size of the message at the time the message is generated.

Notice that the total value carried by a newly-created internal outbound message equals the sum of `value`, `ihr_fee`, and `fwd_fee`. This sum is deducted from the balance of the source account. Of these components, only `value` is always credited to the destination account on message delivery. The `fwd_fee` is collected by the validators on the HR path from the source to the destination, and the `ihr_fee` is either collected by the validators of the destination shardchain (if the message is delivered via IHR), or credited to the destination account.

3.1.9. Code and data portions contained in a message. Apart from the common message information stored in `info`, a message can contain portions of the destination smart contract's code and data. This feature is used, for instance, in the so-called *constructor messages* (cf. **1.7.3**), which are simply internal or inbound external messages with `code` and possibly `data` fields defined in their `init` portions. If the hash of these fields is correct, and the

destination smart contract has no code or data, the values from the message are used instead.[24]

3.1.10. Using `code` and `data` for other purposes. Workchains other than the masterchain and the basic workchain are free to use the trees of cells referred to in the `code`, `data`, and `library` fields for their own purposes. The messaging system itself makes no assumptions about their contents; they become relevant only when a message is processed by a transaction.

3.1.11. Absence of an explicit gas price and gas limit. Notice that messages do not have an explicit gas price and gas limit. Instead, the gas price is set globally by the validators for each workchain (it is a special configurable parameter), and the gas limit for each transaction has also a default value, which is a configurable parameter; the smart contract itself may lower the gas limit during its execution if so desired.

For internal messages, the initial gas limit cannot exceed the Gram value of the message divided by the current gas price. For inbound external messages, the initial gas limit is very small, and the true gas limit is set by the receiving smart contract itself, when it *accepts* the inbound message by the corresponding TVM primitive.

3.1.12. Deserialization of a message payload. The payload, or body, of a message is deserialized by the receiving smart contract when executed by TVM. The messaging system itself makes no assumptions about the internal format of the payload. However, it makes sense to describe the serialization of supported inbound messages by TL or TL-B schemes with 32-bit constructor tags, so that the developers of other smart contracts will know the interface supported by a specific smart contract.

A message is always serialized inside the blockchain as the last field in a cell. Therefore, the blockchain software may assume that whatever bits and references left unparsed after parsing the fields of a `Message` preceding `body` belong to the payload `body` : X, without knowing anything about the serialization of the type X.

[24] More precisely, the information from the `init` field of an inbound message is used either when the receiving account is uninitialized or frozen with the hash of *StateInit* equal to the one expected by the account, or when the receiving account is active, and its code or data is an external hash reference matching the hash of the code or data received in the *StateInit* of the message.

3.1.13. Messages with empty payloads. The payload of a message may happen to be an empty cell slice, having no data bits and no references. By convention, such messages are used for simple value transfers. The receiving smart contract is normally expected to process such messages quietly and to terminate successfully (with a zero exit code), although some smart contracts may perform non-trivial actions even when receiving a message with empty payload. For example, a smart contract may check the resulting balance, and, if it becomes sufficient for a previously postponed action, trigger this action. Alternatively, the smart contract might want to remember in its persistent storage the amount received and the corresponding sender, in order, for instance, to distribute some tokens later to each sender proportionally to the funds transferred.

Notice that even if a smart contract makes no special provisions for messages with empty payloads and throws an exception while processing such messages, the received value (minus the gas payment) will still be added to the balance of the smart contract.

3.1.14. Message source address and logical creation time determine its generating block. Notice that *the source address and the logical creation time of an internal or an outbound external message uniquely determine the block in which the message has been generated*. Indeed, the source address determines the source shardchain, and the blocks of this shardchain are assigned non-intersecting logical time intervals, so only one of them may contain the indicated logical creation time. This is the reason why no explicit mention of the generating block is needed in messages.

3.1.15. Enveloped messages. *Message envelopes* are used for attaching routing information, such as the current (transit) address and the next-hop address, to inbound, transit, and outbound messages (cf. **2.1.16**). The message itself is kept in a separate cell and referred to from the message envelope by a cell reference.

```
interm_addr_regular$0 use_src_bits:(#<= 96)
  = IntermediateAddress;
interm_addr_simple$10 workchain_id:int8 addr_pfx:(64 * Bit)
  = IntermediateAddress;
interm_addr_ext$11 workchain_id:int32 addr_pfx:(64 * Bit)
  = IntermediateAddress;
msg_envelope cur_addr:IntermediateAddress
```

```
next_addr:IntermediateAddress fwd_fee_remaining:Grams
msg:^(Message Any) = MsgEnvelope;
```

The `IntermediateAddress` type is used to describe the intermediate addresses of a message—that is, its current (or transit) address `cur_addr`, and its next-hop address `next_addr`. The first constructor `interm_addr_regular` represents the intermediate address using the optimization described in **2.1.15**, by storing the number of the first bits of the intermediate address that are the same as in the source address; the two other explicitly store the workchain identifier and the first 64 bits of the address inside that workchain (the remaining bits can be taken from the source address). The `fwd_fee_remaining` field is used to explicitly represent the maximum amount of message forwarding fees that can be deducted from the message value during the remaining HR steps; it cannot exceed the value of `fwd_fee` indicated in the message itself.

3.2 Inbound message descriptors

This section discusses *InMsgDescr*, the structure containing a description of all inbound messages imported into a block.[25]

3.2.1. Types and sources of inbound messages. Each inbound message mentioned in *InMsgDescr* is described by a value of type *InMsg* (an "inbound message descriptor"), which specifies the source of the message, the reason for its being imported into this block, and some information about its "fate"—its processing by a transaction or forwarding inside the block.

Inbound messages may be classified as follows:

- *Inbound external messages* — Need no additional reason for being imported into the block, but must be immediately processed by a transaction in the same block.
- *Internal IHR messages with destination addresses in this block* — The reason for their being imported into the block includes a Merkle proof of their generation (i.e., their inclusion in *OutMsgDescr* of their original block). Such a message must be immediately delivered to its final destination and processed by a transaction.

[25]Strictly speaking, *InMsgDescr* is the *type* of this structure; we deliberately use the same notation to describe the only instance of this type in a block.

- *Internal messages with destinations in this block* — The reason for their inclusion is their presence in *OutMsgQueue* of the most recent state of a neighboring shardchain,[26] or their presence in *OutMsgDescr* of this very block. This neighboring shardchain is completely determined by the transit address indicated in the forwarded message envelope, which is replicated in *InMsg* as well. The "fate" of this message is again described by a reference to the processing transaction inside the current block.

- *Immediately routed internal messages* — Essentially a subclass of the previous class of messages. In this case, the imported message is one of the outbound messages generated in this very block.

- *Transit internal messages* — Have the same reason for inclusion as the previous class of messages. However, they are not processed inside the block, but internally forwarded into *OutMsgDescr* and *OutMsgQueue*. This fact, along with a reference to the new envelope of the transit message, must be registered in *InMsg*.

- *Discarded internal messages with destinations in this block* — An internal message with a destination in this block may be imported and immediately discarded instead of being processed by a transaction if it has already been received and processed via IHR in a preceding block of this shardchain. In this case, a reference to the previous processing transaction must be provided.

- *Discarded transit internal messages* — Similarly, a transit message may be discarded immediately after import if it has already been delivered via IHR to its final destination. In this case, a Merkle proof of its processing in the final block (as an IHR message) is required.

3.2.2. Descriptor of an inbound message. Each inbound message is described by an instance of the `InMsg` type, which has six constructors corresponding to the cases listed above in **3.2.1**:

```
msg_import_ext$000 msg:^(Message Any) transaction:^Transaction
              = InMsg;
msg_import_ihr$010 msg:^(Message Any) transaction:^Transaction
```

[26] Recall that a shardchain is considered a neighbor of itself.

```
    ihr_fee:Grams proof_created:^Cell = InMsg;
msg_import_imm$011 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_fin$100 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_tr$101  in_msg:^MsgEnvelope out_msg:^MsgEnvelope
    transit_fee:Grams = InMsg;
msg_discard_fin$110 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams = InMsg;
msg_discard_tr$111 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams proof_delivered:^Cell = InMsg;
```

Notice that the processing transaction is referred to in the first four constructors directly by a cell reference to `Transaction`, even though the logical time of the transaction `transaction_lt:uint64` would suffice for this purpose. Internal consistency conditions ensure that the transaction referred to does belong to the destination smart contract indicated in the message, and that the inbound message processed by that transaction is indeed the one being described in this *InMsg* instance.

Furthermore, notice that `msg_import_imm` could be distinguished from `msg_import_fin` by observing that it is the only case when the logical creation time of the message being processed is greater than or equal to the (minimal) logical time of the block importing the message.

3.2.3. Collecting forwarding and transit fees from imported messages. The *InMsg* structure is also used to indicate the forwarding and transit fees collected from inbound messages. The fee itself is indicated in `ihr_fee`, `fwd_fee`, or `transit_fee` fields; it is absent only in inbound external messages, which use other mechanisms to reward the validators for importing them. The fees must satisfy the following internal consistency conditions:

- For external messages (`msg_import_ext`), there is no forwarding fee.
- For IHR-imported internal messages (`msg_import_ihr`), the fee equals `ihr_fee`, which must coincide with the `ihr_fee` value indicated in the message itself. Notice that `fwd_fee` or `fwd_fee_remaining` are never collected from IHR-imported messages.

- For internal messages delivered to their destination (`msg_import_fin` and `msg_import_imm`), the fee equals the `fwd_fee_remaining` of the enveloped inbound message `in_msg`. Note that it cannot exceed the `fwd_fee` value indicated in the message itself.
- For transit messages (`msg_import_tr`), the fee equals the difference between the `fwd_fee_remaining` values indicated in the `in_msg` and `out_msg` envelopes.
- For discarded messages, the fee also equals the `fwd_fee_remaining` indicated in `in_msg`.

3.2.4. Imported value of an inbound message. Each imported message imports some value—a certain amount of one or more cryptocurrencies—into the block. This imported value is computed as follows:

- An external message imports no value.
- An IHR-imported message imports its `value` plus its `ihr_fee`.
- A delivered or transit internal message imports its `value` plus its `ihr_fee` plus the value of `fwd_fee_remaining` of its `in_msg` envelope.
- A discarded message imports the `fwd_fee_remaining` of its `in_msg` envelope.

Notice that the forwarding and transit fees collected from an imported message do not exceed its imported value.

3.2.5. Augmented hashmaps, or dictionaries. Before continuing, let us discuss the serialization of *augmented* hashmaps, or dictionaries.

Augmented hashmaps are key-value storage structures with n-bit keys and values of some type X, similar to the ordinary hashmaps described in [4, 3.3]. However, each intermediate node of the Patricia tree representing an augmented hashmap is *augmented* by a value of type Y.

These augmentation values must satisfy certain *aggregation conditions*. Typically, Y is an integer type, and the aggregation condition is that the augmentation value of a fork must equal the sum of the augmentation values of its two children. In general, a *fork evaluation function* $S : Y \times Y \to Y$ or $S : Y \to Y \to Y$ is used instead of the sum. The augmentation value of a leaf is usually computed from the value stored in that leaf by means of a *leaf*

evaluation function $L : X \to Y$. The augmentation value of a leaf may be stored explicitly in the leaf along with the value; however, in most cases there is no need for this, because the leaf evaluation function L is very simple.

3.2.6. Serialization of augmented hashmaps. The serialization of augmented hashmaps with n-bit keys, values of type X, and augmentation values of type Y is given by the following TL-B scheme, which is an extension of the one provided in [4, 3.3.3]:

```
ahm_edge#_ {n:#} {X:Type} {Y:Type} {l:#} {m:#}
  label:(HmLabel ~l n) {n = (~m) + l}
  node:(HashmapAugNode m X Y) = HashmapAug n X Y;
ahmn_leaf#_ {X:Type} {Y:Type} extra:Y value:X
  = HashmapAugNode 0 X Y;
ahmn_fork#_ {n:#} {X:Type} {Y:Type}
    left:^(HashmapAug n X Y) right:^(HashmapAug n X Y) extra:Y
  = HashmapAugNode (n + 1) X Y;

ahme_empty$0 {n:#} {X:Type} {Y:Type} extra:Y
          = HashmapAugE n X Y;
ahme_root$1 {n:#} {X:Type} {Y:Type} root:^(HashmapAug n X Y)
  extra:Y = HashmapAugE n X Y;
```

3.2.7. Augmentation of *InMsgDescr*. The collection of inbound message descriptors is augmented by a vector of two currency values, representing the imported value and the forwarding and transit fees collected from a message or a collection of messages:

```
import_fees$_ fees_collected:Grams
  value_imported:CurrencyCollection = ImportFees;
```

3.2.8. Structure of *InMsgDescr*. Now the *InMsgDescr* itself is defined as an augmented hashmap, with 256-bit keys (equal to the representation hashes of imported messages), values of type `InMsg` (cf. **3.2.2**), and augmentation values of type `ImportFees` (cf. **3.2.7**):

```
_ (HashmapAugE 256 InMsg ImportFees) = InMsgDescr;
```

This TL-B notation uses an anonymous constructor `_` to define `InMsgDescr` as a synonym for another type.

3.2.9. Aggregation rules for *InMsgDescr*. The fork evaluation and leaf evaluation functions (cf. **3.2.5**) are not included explicitly in the above notation, because the dependent types of TL-B are not expressive enough for this purpose. In words, the fork evaluation function is just the componentwise addition of two `ImportFees` instances, and the leaf evaluation function is defined by the rules listed in **3.2.3** and **3.2.4**. In this way, the root of the Patricia tree representing an instance of *InMsgDescr* contains an *ImportFees* instance with the total value imported by all inbound messages, and with the total forwarding fees collected from them.

3.3 Outbound message queue and descriptors

This section discusses *OutMsgDescr*, the structure representing all outbound messages of a block, along with their envelopes and brief descriptions of the reasons for including them into *OutMsgDescr*. This structure also describes all modifications of *OutMsgQueue*, which is a part of the shardchain state.

3.3.1. Types of outbound messages. Outbound messages may be classified as follows:

- *External outbound messages*, or "messages to nowhere" — Generated by a transaction inside this block. The reason for including such a message into *OutMsgDescr* is simply a reference to its generating transaction.
- *Immediately processed internal outbound messages* — Generated and processed in this very block, and not included into *OutMsgQueue*. The reason for including such a message is a reference to its generating transaction, and its "fate" is described by a reference to the corresponding entry in *InMsgDescr*.
- *Ordinary (internal) outbound messages* — Generated in this block and included into *OutMsgQueue*.
- *Transit (internal) outbound messages* — Imported into the *InMsgDescr* of the same block and routed via HR until a next-hop address outside the current shard is obtained.

3.3.2. Message dequeueing records. Apart from the above types of outbound messages, *OutMsgDescr* can contain special "message dequeueing

records", which indicate that a message has been removed from the *OutMsgQueue* in this block. The reason for this removal is indicated in the message deletion record; it consists of a reference to the enveloped message being deleted, and of the logical time of the neighboring shardchain block that has this enveloped message in its *InMsgDescr*.

Notice that on some occasions a message may be imported from the *OutMsgQueue* of the current shardchain, internally routed, and then included into *OutMsgDescr* and *OutMsgQueue* again with a different envelope.[27] In this case, a variant of the transit outbound message description is used, which doubles as a message dequeueing record.

3.3.3. Descriptor of an outbound message. Each outbound message is described by an instance of *OutMsg*:

```
msg_export_ext$000 msg:^(Message Any)
    transaction:^Transaction = OutMsg;
msg_export_imm$010 out_msg:^MsgEnvelope
    transaction:^Transaction reimport:^InMsg = OutMsg;
msg_export_new$001 out_msg:^MsgEnvelope
    transaction:^Transaction = OutMsg;
msg_export_tr$011  out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
msg_export_deq$110 out_msg:^MsgEnvelope
    import_block_lt:uint64 = OutMsg;
msg_export_tr_req$111 out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
```

The last two descriptions have the effect of removing (dequeueing) the message from *OutMsgQueue* instead of inserting it. The last one re-inserts the message into *OutMsgQueue* with a new envelope after performing the internal routing (cf. **2.1.11**).

3.3.4. Exported value of an outbound message. Each outbound message described by an *OutMsg* exports some value—a certain amount of one or more cryptocurrencies—from the block. This exported value is computed as follows:

[27] This situation is rare and occurs only after shardchain merge events. Normally the messages imported from the *OutMsgQueue* of the same shardchain have destinations inside this shardchain, and are processed accordingly instead of being re-queued.

- An external outbound message exports no value.
- An internal message, generated in this block, exports its `value` plus its `ihr_fee` plus its `fwd_fee`. Notice that `fwd_fee` must be equal to the `fwd_fee_remaining` indicated in the `out_msg` envelope.
- A transit message exports its `value` plus its `ihr_fee` plus the value of `fwd_fee_remaining` of its `out_msg` envelope.
- The same holds for `msg_export_tr_req`, the constructor of *OutMsg* used for re-inserted dequeued messages.
- A message dequeueing record (`msg_export_deq`; cf. **3.3.2**) exports no value.

3.3.5. Structure of *OutMsgDescr*. The *OutMsgDescr* itself is simply an augmented hashmap (cf. **3.2.5**), with 256-bit keys (equal to the representation hash of the message), values of type *OutMsg*, and augmentation values of type *CurrencyCollection*:

```
_ (HashmapAugE 256 OutMsg CurrencyCollection) = OutMsgDescr;
```

The augmentation is the *exported value* of the corresponding message, aggregated by means of the sum, and computed at the leaves as explained in **3.3.4**. In this way, the total exported value appears near the root of the Patricia tree representing *OutMsgDescr*.

The most important consistency condition for *OutMsgDescr* is that its entry with key k must be an *OutMsg* describing a message m with representation hash $\text{HASH}^\flat(m) = k$.

3.3.6. Structure of *OutMsgQueue*. Recall (cf. **1.2.7**) that *OutMsgQueue* is a part of the blockchain state, not of a block. Therefore, a block contains only hash references to its initial and final state, and its newly-created cells.

The structure of *OutMsgQueue* is simple: it is just an augmented hashmap with 352-bit keys and values of type *OutMsg*:

```
_ (HashmapAugE 352 OutMsg uint64) = OutMsgQueue;
```

The key used for an outbound message m is the concatenation of its 32-bit next-hop *workchain_id*, the first 64 bits of the next-hop address inside that workchain, and the representation hash $\text{HASH}^\flat(m)$ of the message m itself.

The augmentation is by the logical creation time $\text{Lt}(m)$ of message m at the leaves, and by the minimum of the augmentation values of the children at the forks.

The most important consistency condition for *OutMsgQueue* is that the value at key k must indeed contain an enveloped message with the expected next-hop address and representation hash.

3.3.7. Consistency conditions for *OutMsg*. Several internal consistency conditions are imposed on *OutMsg* instances present in *OutMsgDescr*. They include the following:

- Each of the first three constructors of outbound message descriptions includes a reference to the generating transaction. This transaction must belong to the source account of the message, it must contain a reference to the specified message as one of its outbound messages, and it must be recoverable by looking it up by its `account_id` and `transaction_id`.
- `msg_export_tr` and `msg_export_tr_req` must refer to an *InMsg* instance describing the same message (in a different original envelope).
- If one of the first four constructors is used, the message must be absent in the initial *OutMsgQueue* of the block; otherwise, it must be present.
- If `msg_export_deq` is used, the message must be absent in the final *OutMsgQueue* of the block; otherwise, it must be present.
- If a message is not mentioned in *OutMsgDescr*, it must be the same in the initial and final *OutMsgQueue*s of the block.

4 Accounts and transactions

This chapter discusses the layout of *accounts* (or *smart contracts*) and their *state* in the TON Blockchain. It also considers *transactions*, which are the only way to modify the state of an account, and to process inbound messages and generate new outbound messages.

4.1 Accounts and their states

Recall that a *smart contract* and an *account* are the same thing in the context of the TON Blockchain, and that these terms can be used interchangeably, at least as long as only small (or "usual") smart contracts are considered. A *large* smart contract may employ several accounts lying in different shardchains of the same workchain for load balancing purposes.

An account is *identified* by its full address, and is *completely described* by its state. In other words, there is nothing else in an account apart from its address and state.

4.1.1. Account addresses. In general, an account is completely identified by its *full address*, consisting of a 32-bit *workchain_id*, and the (usually 256-bit) *internal address* or *account identifier account_id* inside the chosen workchain. In the basic workchain ($workchain\_id = 0$) and in the masterchain ($workchain\_id = -1$) the internal address is always 256-bit. In these workchains,[28] *account_id* cannot be chosen arbitrarily, but must be equal to the hash of the initial code and data of the smart contract; otherwise, it will be impossible to initialize the account with the intended code and data (cf. **1.7.3**), and to do anything with the accumulated funds in the account balance.

4.1.2. Zero account. By convention, the *zero account* or *account with zero address* accumulates the processing, forwarding, and transit fees, as well as any other payments collected by the validators of the masterchain or a workchain. Furthermore, the zero account is a "large smart contract", meaning that each shardchain has its instance of the zero account, with the most significant bits of the address adjusted to lie in the shard. Any funds transferred to the zero account, intentionally or by accident, are effectively

[28] For simplicity, we sometimes treat the masterchain as just another workchain with $workchain\_id = -1$.

a gift for the validators. For example, a smart contract might destroy itself by sending all its funds to the zero account.

4.1.3. Small and large smart contracts. By default, smart contracts are "small", meaning that they have one account address belonging to exactly one shardchain at any given moment of time. However, one can create a "large smart contract of splitting depth d", meaning that up to 2^d instances of the smart contract may be created, with the first d bits of the original address of the smart contract replaced by arbitrary bit sequences.[29] One can send messages to such smart contracts using internal anycast addresses with `anycast` set to d (cf. **3.1.2**). Furthermore, the instances of the large smart contract are allowed to use this anycast address as the source address of their generated messages.

An instance of a large smart contract is an account with non-zero *maximal splitting depth* d.

4.1.4. The three kinds of accounts. There are three kinds of accounts:

- *Uninitialized* — The account only has a balance; its code and data have not yet been initialized.
- *Active* — The account's code and data have been initialized as well.
- *Frozen* — The account's code and data have been replaced by a hash, but the balance is still stored explicitly. The balance of a frozen account may effectively become negative, reflecting due storage payments.

4.1.5. Storage profile of an account. The *storage profile* of an account is a data structure describing the amount of persistent blockchain state storage used by that account. It describes the total amount of cells, data bits, and internal and external cell references used.

```
storage_used$_ cells:(VarUInteger 7) bits:(VarUInteger 7)
  ext_refs:(VarUInteger 7) int_refs:(VarUInteger 7)
  public_cells:(VarUInteger 7) = StorageUsed;
```

[29] In fact, up to the first d bits are replaced in such a way that each shard contains at most one instance of the large smart contract, and that shards (w, s) with prefix s of length $|s| \leq d$ contain exactly one instance.

The same type `StorageUsed` may represent the storage profile of a message, as required, for instance, to compute `fwd_fee`, the total forwarding fee for Hypercube Routing. The storage profile of an account has some additional fields indicating the last time when the storage fees were exacted:

```
storage_info$_ used:StorageUsed last_paid:uint32
              due_payment:(Maybe Grams) = StorageInfo;
```

The `last_paid` field contains either the unixtime of the most recent storage payment collected (usually this is the unixtime of the most recent transaction), or the unixtime when the account was created (again, by a transaction). The `due_payment` field, if present, accumulates the storage payments that could not be exacted from the balance of the account, represented by a strictly positive amount of nanograms; it can be present only for uninitialized or frozen accounts that have a balance of zero Grams (but may have non-zero balances in other cryptocurrencies). When `due_payment` becomes larger than the value of a configurable parameter of the blockchain, the account is destroyed altogether, and its balance, if any, is transferred to the zero account.

4.1.6. Account description. The state of an account is represented by an instance of type *Account*, described by the following TL-B scheme:[30]

```
account_none$0 = Account;
account$1 addr:MsgAddressInt storage_stat:StorageInfo
          storage:AccountStorage = Account;

account_storage$_ last_trans_lt:uint64
    balance:CurrencyCollection state:AccountState
  = AccountStorage;

account_uninit$00 = AccountState;
account_active$1 _:StateInit = AccountState;
account_frozen$01 state_hash:uint256 = AccountState;

acc_state_uninit$00 = AccountStatus;
acc_state_frozen$01 = AccountStatus;
```

[30]This scheme uses anonymous constructors and anonymous fields, both represented by an underscore `_`.

```
acc_state_active$10 = AccountStatus;
acc_state_nonexist$11 = AccountStatus;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

Notice that `account_frozen` contains the representation hash of an instance of *StateInit*, instead of that instance itself, which would otherwise be contained in an `account_active`; `account_uninit` is similar to `account_frozen`, but it does not contain an explicit `state_hash`, because it is assumed to be equal to the internal address of the account (*account_id*), already present in the `addr` field. The `split_depth` field is present and non-zero only in instances of large smart contracts. The `special` field may be present only in the masterchain—and within the masterchain, only in some *fundamental* smart contracts required for the whole system to function.

The storage statistics kept in `storage_stat` reflect the total storage usage of cell slice `storage`. In particular, the bits and cells used to store the `balance` are also reflected in `storage_stat`.

When a non-existent account needs to be represented, the `account_none` constructor is used.

4.1.7. Account state as a message from an account to its future self. Notice that the account state is very similar to a message sent from an account to its future self participating in the next transaction, for the following reasons:

- The account state does not change between two consecutive transactions of the same account, so it is completely similar in this respect to a message sent from the earlier transaction to the later one.
- When a transaction is processed, its inputs are an inbound message and the previous account state; its outputs are outbound messages generated and the next account state. If we treat the state as a special kind of message, we see that every transaction has exactly two inputs (the account state and an inbound message) and at least one output.

- Both a message and the account state can carry code and data in an instance of *StateInit*, and some value in their `balance`.
- An account is initialized by a *constructor message*, which essentially carries the future state and balance of the account.
- On some occasions messages are converted into account states, and vice versa. For instance, when a shardchain merge event occurs, and two accounts that are instances of the same large contract need to be merged, one of them is converted into a message sent to the other one (cf. **4.2.11**). Similarly, when a shardchain split event occurs, and an instance of a large smart contract needs to be split into two, this is achieved by a special transaction that creates the new instance by means of a constructor message sent from the previously existing instance to the new one (cf. **4.2.10**).
- One may say that a message is involved in transferring some information *across space* (between different shardchains, or at least accountchains), while an account state transfers information *across time* (from the past to the future of the same account).

4.1.8. Differences between messages and account states. Of course, there are important differences, too. For example:

- The account state is transferred only "in time" (for a shardchain block to its successor), but never "in space" (from one shardchain to another). As a consequence, this transfer is done implicitly, without creating complete copies of the account state anywhere in the blockchain.
- Storage payments collected by the validators for keeping the account state usually are considerably smaller than message forwarding fees for the same amount of data.
- When an inbound message is delivered to an account, it is the code from the account that is invoked, not the code from the message.

4.1.9. The combined state of all accounts in a shard. The split part of the shardchain state (cf. **1.2.1** and **1.2.2**) is given by

```
_ (HashmapAugE 256 Account CurrencyCollection) = ShardAccounts;
```

This is simply a dictionary with 256-bit *account_id*s as keys and corresponding account states as values, sum-augmented by the balances of the accounts. In this way the sum of balances of all accounts in a shardchain is computed, so that one can easily check the total amount of cryptocurrency "stored" in a shard.

Internal consistency conditions ensure that the address of an account referred to by key k in *SmartAccounts* is indeed equal to k. An additional internal consistency condition requires that all keys k begin with the shard prefix s.

4.1.10. Account owner and interface descriptions. One may want to include some optional information in a controlled account. For example, an individual user or a company may want to add a text description field to their wallet account, with the user's or company's name or address (or their hash, if the information should not be made publicly available). Alternatively, a smart contract may offer a machine-readable or human-readable description of its supported methods and their intended application, which might be used by advanced wallet applications to construct drop-down menus and forms helping a human user to create valid messages to be sent to that smart contract.

One way of including such information is to reserve, say, the second reference in the `data` cell of the state of an account for a dictionary with 64-bit keys (corresponding to some identifiers of the standard types of extra data one might want to store) and corresponding values. Then a blockchain explorer would be able to extract the required value, along with a Merkle proof if necessary.

A better way of doing this is by defining some *get methods* in the smart contract.

4.1.11. Get methods of a smart contract. *Get methods* are executed by a stand-alone instance of TVM with the account's code and data loaded into it. The required parameters are passed on the stack (say, a magic number indicating the field to be fetched or the specific get method to be invoked), and the results are returned on the TVM stack as well (say, a cell slice containing the serialization of a string with the account owner's name).

As a bonus, get methods may be used to get answers to more sophisticated queries than just fetching a constant object. For instance, TON DNS registry smart contracts provide get methods to look up a domain string in the registry and return the corresponding record, if found.

By convention, get methods use large *negative* 32-bit or 64-bit indices or magic numbers, and internal functions of a smart contract use consecutive *positive* indices, to be used in TVM's `CALLDICT` instruction. The `main()` function of a smart contract, used to process inbound messages in ordinary transactions, always has index zero.

4.2 Transactions

According to the Infinite Sharding Paradigm and the actor model, the three principal components of the TON Blockchain are *accounts* (along with their states), *messages,* and *transactions*. Previous sections have already discussed the first two; this section considers transactions.

In contrast with messages, which have essentially the same headers throughout all workchains of the TON Blockchain, and accounts, which have at least some common parts (the address and the balance), our discussion of transactions is necessarily limited to the masterchain and the basic workchain. Other workchains may define completely different kinds of transactions.

4.2.1. Logical time of a transaction. Each transaction t has a logical time interval $\text{Lt}^\bullet(t) = [\text{Lt}^-(t), \text{Lt}^+(t))$ assigned to it (cf. **1.4.6** and **1.4.3**). By convention, a transaction t generating n outbound messages $m_1, \ldots, m_n$ is assigned a logical time interval of length $n + 1$, so that

$$\text{Lt}^+(t) = \text{Lt}^-(t) + n + 1 \quad . \tag{16}$$

We also set $\text{Lt}(t) := \text{Lt}^-(t)$, and assign the logical creation time of message m_i, where $1 \leq i \leq n$, by

$$\text{Lt}(m_i) = \text{Lt}^-(m_i) := \text{Lt}^-(t) + i, \quad \text{Lt}^+(m_i) := \text{Lt}^-(m_i) + 1 \quad . \tag{17}$$

In this way, each generated outbound message is assigned its own unit interval inside the logical time interval $\text{Lt}^\bullet(t)$ of transaction t.

4.2.2. Logical time uniquely identifies transactions and outbound messages of an account. Recall that the conditions imposed on logical time imply that $\text{Lt}^-(t) \geq \text{Lt}^+(t')$ for any preceding transaction t' of the same account ξ, and that $\text{Lt}^-(t) > \text{Lt}(m)$ if m is the inbound message processed by transaction t. In this way, the logical time intervals of transactions of the same account do not intersect each other, and as a consequence, the logical time intervals of all outbound messages generated by an account do

not intersect each other either. In other words, all $\text{Lt}(m)$ are different, when m runs through all outbound messages of the same account ξ.

In this way, $\text{Lt}(t)$ and $\text{Lt}(m)$, when combined with an account identifier ξ, uniquely determine a transaction t or an outbound message m of that account. Furthermore, if one has an ordered list of all transactions of an account along with their logical times, it is easy to find the transaction that generated a given outbound message m, simply by looking up the transaction t with logical time $\text{Lt}(t)$ nearest to $\text{Lt}(m)$ from below.

4.2.3. Generic components of a transaction. Each transaction t contains or indirectly refers to the following data:

- The account ξ to which the transaction belongs.
- The logical time $\text{Lt}(t)$ of the transaction.
- One or zero inbound messages m processed by the transaction.
- The number of generated outbound messages $n \geq 0$.
- The outbound messages $m_1, \ldots, m_n$.
- The initial state of account ξ (including its balance).
- The final state of account ξ (including its balance).
- The total fees collected by the validators.
- A detailed description of the transaction containing all or some data needed to validate it, including the kind of the transaction (cf. **4.2.4**) and some of the intermediate steps performed.

Of these components, all but the very last one are quite general and might appear in other workchains as well.

4.2.4. Kinds of transactions. There are different kinds of transactions allowed in the masterchain and the shardchains. *Ordinary* transactions consist in the delivery of one inbound message to an account, and its processing by that account's code; this is the most common kind of transaction. Additionally, there are several kinds of *exotic* transactions.

Altogether, there are six kinds of transactions:

- *Ordinary transactions* — Belong to an account ξ. They process exactly one inbound message m (described in *InMsgDescr* of the encompassing block) with destination ξ, compute the new state of the account, and generate several outbound messages (registered in *OutMsgDescr*) with source ξ.

- *Storage transactions* — Can be inserted by validators at their discretion. They do not process any inbound message and do not invoke any code. Their only effect is to collect storage payments from an account, affecting its storage statistics and its balance. If the resulting Gram balance of the account becomes less than a certain amount, the account may be frozen and its code and data replaced by their combined hash.

- *Tick transactions* — Automatically invoked for certain special accounts (smart contracts) in the masterchain that have the `tick` flag set in their state, as the very first transactions in every masterchain block. They have no inbound message, but may generate outbound messages and change the account state. For instance, *validator elections* are performed by tick transactions of special smart contracts in the masterchain.

- *Tock transactions* — Similarly automatically invoked as the very last transactions in every masterchain block for certain special accounts.

- *Split transactions* — Invoked as the last transactions of shardchain blocks immediately preceding a shardchain split event. They are triggered automatically for instances of large smart contracts that need to produce a new instance after splitting.

- *Merge transactions* — Similarly invoked as the first transactions of shardchain blocks immediately after a shardchain merge event, if an instance of a large smart contract needs to be merged with another instance of the same smart contract.

Notice that out of these six kinds of transactions, only four can occur in the masterchain, and another subset of four can occur in the basic workchain.

4.2.5. Phases of an ordinary transaction. An ordinary transaction is performed in several *phases*, which may be thought of as several "subtransactions" tightly bound into one:

- *Storage phase* — Collects due storage payments for the account state (including smart-contract code and data, if present) up to the present time. The smart contract may be *frozen* as a result. If the smart contract did not exist before, the storage phase is absent.

- *Credit phase* — The account is credited with the value of the inbound message received.

- *Computing phase* — The code of the smart contract is invoked inside an instance of TVM with adequate parameters, including a copy of the inbound message and of the persistent data, and terminates with an exit code, the new persistent data, and an *action list* (which includes, for instance, outbound messages to be sent). The processing phase may lead to the creation of a new account (uninitialized or active), or to the activation of a previously uninitialized or frozen account. The *gas payment*, equal to the product of the gas price and the gas consumed, is exacted from the account balance.

- *Action phase* — If the smart contract has terminated successfully (with exit code 0 or 1), the actions from the list are performed. If it is impossible to perform all of them—for example, because of insufficient funds to transfer with an outbound message—then the transaction is aborted and the account state is rolled back. The transaction is also aborted if the smart contract did not terminate successfully, or if it was not possible to invoke the smart contract at all because it is uninitialized or frozen.

- *Bounce phase* — If the transaction has been aborted, and the inbound message has its `bounce` flag set, then it is "bounced" by automatically generating an outbound message (with the `bounce` flag clear) to its original sender. Almost all value of the original inbound message (minus gas payments and forwarding fees) is transferred to the generated message, which otherwise has an empty body.

4.2.6. Bouncing inbound messages to non-existent accounts. Notice that if an inbound message with its `bounce` flag set is sent to a previously non-existent account, and the transaction is aborted (for instance, because there is no code and data with the correct hash in the inbound message, so the virtual machine could not be invoked at all), then the account is not

created even as an uninitialized account, since it would have zero balance and no code and data anyways.[31]

4.2.7. Processing of an inbound message is split between computing and action phases. Notice that the processing of an inbound message is in fact split into two phases: the *computing* phase and the *action* phase. During the computing phase, the virtual machine is invoked and the necessary computations are performed, but no actions outside the virtual machine are taken. In other words, *the execution of a smart contract in TVM has no side effects*; there is no way for a smart contract to interact with the blockchain directly during its execution. Instead, TVM primitives such as `SENDMSG` simply store the required action (e.g., the outbound message to be sent) into the action list being gradually accumulated in TVM control register `c5`. The actions themselves are postponed until the action phase, during which the user smart contract is not invoked at all.

4.2.8. Reasons for splitting the processing into computation and action phases. Some reasons for such an arrangement are:

- It is simpler to abort the transaction if the smart contract eventually terminates with an exit code other than 0 or 1.
- The rules for processing output actions may be changed without modifying the virtual machine. (For instance, new output actions may be introduced.)
- The virtual machine itself may be modified or even replaced by another one (for instance, in a new workchain) without changing the rules for processing output actions.
- The execution of the smart contract inside the virtual machine is completely isolated from the blockchain and is a *pure computation*. As a consequence, this execution may be *virtualized* inside the virtual machine itself by means of TVM's `RUNVM` primitive, a useful feature for validator smart contracts and for smart contracts controlling payment

[31] In particular, if a user mistakenly sends some funds to a non-existent address in a bounceable message, the funds will not be wasted, but rather will be returned (bounced) back. Therefore, a user wallet application should set the `bounce` flag in all generated messages by default unless explicitly instructed otherwise. However, non-bounceable messages are indispensable in some situations (cf. **1.7.6**).

channels and other sidechains. Additionally, the virtual machine may be *emulated* inside itself or a stripped-down version of itself, a useful feature for validating the execution of smart contracts inside TVM.[32]

4.2.9. Storage, tick, and tock transactions. Storage transactions are very similar to a stand-alone storage phase of an ordinary transaction. Tick and tock transactions are similar to ordinary transactions without credit and bounce phases, because there is no inbound message.

4.2.10. Split transactions. Split transactions in fact consist of two transactions. If an account ξ needs to be split into two accounts ξ and ξ':

- First a *split prepare transaction*, similar to a tock transaction (but in a shardchain instead of the masterchain), is issued for account ξ. It must be the last transaction for ξ in a shardchain block. The output of the processing stage of a split prepare transaction consists not only of the new state of account ξ, but also of the new state of account ξ', represented by a constructor message to ξ' (cf. **4.1.7**).
- Then a *split install transaction* is added for account ξ', with a reference to the corresponding split prepare transaction. The split install transaction must be the only transaction for a previously non-existent account ξ' in the block. It effectively sets the state of ξ' as defined by the split prepare transaction.

4.2.11. Merge transactions. Merge transactions also consist of two transactions each. If an account ξ' needs to be merged into account ξ:

- First a *merge prepare transaction* is issued for ξ', which converts all of its persistent state and balance into a special constructor message with destination ξ (cf. **4.1.7**).
- Then a *merge install transaction* for ξ, referring to the corresponding merge prepare transaction, processes that constructor message. The merge install transaction is similar to a tick transaction in that it must be the first transaction for ξ in a block, but it is located in a shardchain block, not in the masterchain, and it has a special inbound message.

[32] A reference implementation of a TVM emulator running in a stripped-down version of TVM may be committed into the masterchain to be used when a disagreement between the validators on a specific run of TVM arises. In this way, flawed implementations of TVM may be detected. The reference implementation then serves as an authoritative source on the operational semantics of TVM. (Cf. [4, B.2])

4.2.12. Serialization of a general transaction. Any transaction contains the fields listed in **4.2.3**. As a consequence, there are some common components in all transactions:

```
transaction$_ account_addr:uint256 lt:uint64 outmsg_cnt:uint15
  orig_status:AccountStatus end_status:AccountStatus
  in_msg:(Maybe ^(Message Any))
  out_msgs:(HashmapE 15 ^(Message Any))
  total_fees:Grams state_update:^(MERKLE_UPDATE Account)
  description:^TransactionDescr = Transaction;

!merkle_update#02 {X:Type} old_hash:uint256 new_hash:uint256
  old:^X new:^X = MERKLE_UPDATE X;
```

The exclamation mark in the TL-B declaration of a `merkle_update` indicates special processing required for such values. In particular, they must be kept in a separate cell, which must be marked as *exotic* by a bit in its header (cf. [4, 3.1]).

A full explanation of the serialization of *TransactionDescr*, which describes one transaction according to its kind listed in **4.2.4**, can be found in **4.3**.

4.2.13. Representation of outbound messages generated by a transaction. The outbound messages generated by a transaction t are kept in a dictionary `out_msgs` with 15-bit keys equal to 0, 1, ..., $n-1$, where $n =$ `outmsg_cnt` is the number of generated outbound messages. Message m_{i+1} with index $0 \le i < n$ must have $\text{LT}(m_{i+1}) = \text{LT}(t) + i + 1$, and $\text{LT}(t) = \text{LT}^{-}(t)$ is explicitly stored in the `lt` field.

4.2.14. Consistency conditions for transactions. The common serialization of the fields present in a *Transaction*, independent of its type and description, enables us to impose several "external" consistency conditions on any transaction. The most important of them involves the *value flow* inside the transaction: the sum of all inputs (the import value of the inbound message plus the original balance of the account) must equal the sum of all outputs (the resulting balance of the account, plus the sum of the export values of all outbound messages, plus all storage, processing, and forwarding fees collected by the validators). In this way, a surface inspection of a transaction, which processes an inbound message with an import value of 1 Gram

received by an account with an initial balance of 10 Grams, generating an outbound message with an export value of 100 Grams in the process, will reveal its invalidity even before checking all the details of the TVM execution.

Other consistency conditions may slightly depend on the description of the transaction. For instance, the inbound message processed by an ordinary transaction must be registered in the *InMsgDescr* of the encompassing block, and the corresponding record must contain a reference to this transaction. Similarly, all outbound messages generated by all transactions (with the exception of one special message generated by a split prepare or merge prepare transaction) must be registered in *OutMsgDescr*.

4.2.15. Collection of all transactions of an account. All transactions in a block belonging to the same account ξ are collected into an "accountchain block" *AccountBlock,* which essentially is a dictionary `transactions` with 64-bit keys, each equal to the logical time of the corresponding transaction:

```
acc_trans$_ account_addr:uint256
            transactions:(HashmapAug 64 ^Transaction Grams)
            state_update:^(MERKLE_UPDATE Account)
          = AccountBlock;
```

The `transactions` dictionary is sum-augmented by a *Grams* value, which aggregates the total fees collected from these transactions.

In addition to this dictionary, an *AccountBlock* contains a Merkle update (cf. [4, 3.1]) of the total state of the account. If an account did not exist before the block, its state is represented by an `account_none`.

4.2.16. Consistency conditions for *AccountBlock*s. There are several general consistency conditions imposed on an *AccountBlock*. In particular:

- The transaction appearing as a value in the augmented `transactions` dictionary must have its `lt` value equal to its key.
- All transactions must belong to an account whose address `account_addr` is indicated in the *AccountBlock*.
- If t and t' are two transactions with $\text{Lt}(t) < \text{Lt}(t')$, and their keys are consecutive in `transactions`, meaning that there is no transaction t'' with $\text{Lt}(t) < \text{Lt}(t'') < \text{Lt}(t')$, then the final state of t must correspond to the initial state of t' (their hashes as explicitly indicated in the Merkle updates must be equal).

- If t is the transaction with minimal $\text{Lt}(t)$, its initial state must coincide with the initial state as indicated in `state_update` of the *AccountBlock*.
- If t is the transaction with maximal $\text{Lt}(t)$, its final state must coincide with the final state as indicated in `state_update` of the *AccountBlock*.
- The list of transactions must be non-empty.

These conditions simply express the fact that the state of an account may change only as the result of performing a transaction.

4.2.17. Collection of all transactions in a block. All transactions in a block are represented by (cf. **1.2.1**):

```
_ (HashmapAugE 256 AccountBlock Grams) = ShardAccountBlocks;
```

4.2.18. Consistency conditions for the collection of all transactions. Again, consistency conditions are imposed on this structure, requiring that the value at key ξ be an *AccountBlock* with address equal to ξ. Further consistency conditions relate this structure with the initial and final states of the shardchain indicated in the block, requiring that:

- If *ShardAccountBlock* has no key ξ, then the state of account ξ in the initial and in the final state of the block must coincide (or it must be absent from both).
- If ξ is present in *ShardAccountBlock*, its initial and final states as indicated in *AccountBlock* must match those indicated in the initial and final states of the shardchain block, expressed by instances of *ShardAccounts* (cf. **4.1.9**).

These conditions express that the shardchain state is indeed composed out of the states of separate accountchains.

4.3 Transaction descriptions

This section presents the specific TL-B schemes for transaction descriptions according to the classification provided in **4.2.4**.

4.3.1. Reasons for omitting data from a transaction description. A transaction description for a blockchain featuring a Turing-complete virtual machine for smart-contract execution is necessarily incomplete. Indeed, a truly complete description would contain all the intermediate states of the virtual machine after each instruction is executed, something that cannot fit into a blockchain block of a reasonable size. Therefore, the description of such a transaction is likely to contain only the total number of steps and the hashes of the initial and final states of the virtual machine. The validation of such a transaction will necessarily involve the execution of the smart contract to reproduce all the intermediate steps and the final result.

If we compress the sequence of all intermediate steps of the virtual machine into just the hashes of the initial and final states, then no transaction details at all need to be included: a validator able to check the execution of the virtual machine by itself would also be able to check all the other actions of the transaction starting from its initial data without these details.

4.3.2. Reasons for including data into a transaction description. The above considerations notwithstanding, there are still several reasons to introduce some details in the transaction description:

- We want to impose external consistency conditions on the transaction, so that at least the validity of the value flow inside the transaction and the validity of inbound and outbound messages can be quickly checked without invoking the virtual machine (cf. **4.2.14**). This at least guarantees the invariance of the total amount of each cryptocurrency in the blockchain, even if it does not guarantee the correctness of its distribution.

- We want to be able to trace principal state changes of an account (such as its being created, activated, or frozen) by inspecting the data stored in the transaction description, without figuring out the missing details of the transaction. This simplifies the verification of the consistency conditions between the accountchain and shardchain states in a block.

- Finally, certain information—such as the total steps of the virtual machine, the hashes of its initial and final states, the total gas consumed, and the exit code—might considerably simplify the debugging and implementation of the TON Blockchain software. (This information would help a human programmer understand what has happened in a particular blockchain block.)

On the other hand, we want to minimize the size of each transaction, because we want to maximize the number of transactions that can fit into each (bounded-size) block. Therefore, all information not required for one of the above reasons is omitted.

4.3.3. Description of a storage phase. The storage phase is present in several kinds of transactions, so a common representation for this phase is used:

```
tr_phase_storage$_ storage_fees_collected:Grams
  storage_fees_due:(Maybe Grams)
  status_change:AccStatusChange
  = TrStoragePhase;

acst_unchanged$0 = AccStatusChange;  // x -> x
acst_frozen$10 = AccStatusChange;    // init -> frozen
acst_deleted$11 = AccStatusChange;   // frozen -> deleted
```

4.3.4. Description of a credit phase. The credit phase can result in the collection of some due payments:

```
tr_phase_credit$_ due_fees_collected:(Maybe Grams)
  credit:CurrencyCollection = TrCreditPhase;
```

The sum of `due_fees_collected` and `credit` must equal the value of the message received, plus its `ihr_fee` if the message has not been received via IHR (otherwise the `ihr_fee` is awarded to the validators).

4.3.5. Description of a computing phase. The computing phase consists in invoking TVM with correct inputs. On some occasions, TVM cannot be invoked at all (e.g., if the account is absent, not initialized, or frozen, and the inbound message being processed has no code or data fields or these fields have an incorrect hash); this is reflected by corresponding constructors.

```
tr_phase_compute_skipped$0 reason:ComputeSkipReason
  = TrComputePhase;
tr_phase_compute_vm$1 success:Bool msg_state_used:Bool
  account_activated:Bool gas_fees:Grams
  _:^[ gas_used:(VarUInteger 7)
  gas_limit:(VarUInteger 7) gas_credit:(Maybe (VarUInteger 3))
```

```
  mode:int8 exit_code:int32 exit_arg:(Maybe int32)
  vm_steps:uint32
  vm_init_state_hash:uint256 vm_final_state_hash:uint256 ]
  = TrComputePhase;
cskip_no_state$00 = ComputeSkipReason;
cskip_bad_state$01 = ComputeSkipReason;
cskip_no_gas$10 = ComputeSkipReason;
```

The TL-B construct `_:^[...]` describes a reference to a cell containing the fields listed inside the square brackets. In this way, several fields can be moved from a cell containing a large record into a separate subcell.

4.3.6. Skipped computing phase. If the computing phase has been skipped, possible reasons include:

- The absence of funds to buy gas.
- The absence of a state (i.e., smart-contract code and data) in both the account (non-existing, uninitialized, or frozen) and the message.
- An invalid state passed in the message (i.e., the state's hash differs from the expected value) to a frozen or uninitialized account.

4.3.7. Valid computing phase. If there is no reason to skip the computing phase, TVM is invoked and the results of the computation are logged. Possible parameters are as follows:

- The `success` flag is set if and only if `exit_code` is either 0 or 1.
- The `msg_state_used` parameter reflects whether the state passed in the message has been used. If it is set, the `account_activated` flag reflects whether this has resulted in the activation of a previously frozen, uninitialized or non-existent account.
- The `gas_fees` parameter reflects the total gas fees collected by the validators for executing this transaction. It must be equal to the product of `gas_used` and `gas_price` from the current block header.
- The `gas_limit` parameter reflects the gas limit for this instance of TVM. It equals the lesser of either the Grams credited in the credit phase from the value of the inbound message divided by the current gas price, or the global per-transaction gas limit.

- The `gas_credit` parameter may be non-zero only for external inbound messages. It is the lesser of either the amount of gas that can be paid from the account balance or the maximum gas credit.
- The `exit_code` and `exit_args` parameters represent the status values returned by TVM.
- The `vm_init_state_hash` and `vm_final_state_hash` parameters are the representation hashes of the original and resulting states of TVM, and `vm_steps` is the total number of steps performed by TVM (usually equal to two plus the number of instructions executed, including implicit `RET`s).[33]

4.3.8. Description of the action phase. The action phase occurs after a valid computation phase. It attempts to perform the actions stored by TVM during the computing phase into the *action list*. It may fail, because the action list may turn out to be too long, contain invalid actions, or contain actions that cannot be completed (for instance, because of insufficient funds to create an outbound message with the required value).

```
tr_phase_action$_ success:Bool valid:Bool no_funds:Bool
  status_change:AccStatusChange
  total_fwd_fees:(Maybe Grams) total_action_fees:(Maybe Grams)
  result_code:int32 result_arg:(Maybe int32) tot_actions:int16
  spec_actions:int16 msgs_created:int16
  action_list_hash:uint256 tot_msg_size:StorageUsed
  = TrActionPhase;
```

4.3.9. Description of the bounce phase.

```
tr_phase_bounce_negfunds$00 = TrBouncePhase;
tr_phase_bounce_nofunds$01 msg_size:StorageUsed
  req_fwd_fees:Grams = TrBouncePhase;
tr_phase_bounce_ok$1 msg_size:StorageUsed
  msg_fees:Grams fwd_fees:Grams = TrBouncePhase;
```

4.3.10. Description of an ordinary transaction.

[33]Notice that this record does not represent a change in the state of the account, because the transaction may still be aborted during the action phase. In that case, the new persistent data indirectly referenced by `vm_final_state_hash` will be discarded.

```
trans_ord$0000 storage_ph:(Maybe TrStoragePhase)
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool bounce:(Maybe TrBouncePhase)
  destroyed:Bool
  = TransactionDescr;
```

Several consistency conditions are imposed on this structure:

- `action` is absent if and only if the computing phase was unsuccessful.
- The `aborted` flag is set either if there is no action phase or if the action phase was unsuccessful.
- The bounce phase occurs only if the `aborted` flag is set and the inbound message was bounceable.

4.3.11. Description of a storage transaction. A storage transaction consists just of one stand-alone storage phase:

```
trans_storage$0001 storage_ph:TrStoragePhase
  = TransactionDescr;
```

4.3.12. Description of tick and tock transactions. Tick and tock transactions are similar to ordinary transactions without an inbound message, so there are no credit or bounce phases:

```
trans_tick_tock$001 is_tock:Bool storage:TrStoragePhase
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool = TransactionDescr;
```

4.3.13. Split prepare and install transactions. A split prepare transaction is similar to a tock transaction in a masterchain, but it must generate exactly one special constructor message; otherwise, the action phase is aborted.

```
split_merge_info$_ cur_shard_pfx_len:(## 6)
  acc_split_depth:(## 6) this_addr:uint256 sibling_addr:uint256
  = SplitMergeInfo;
trans_split_prepare$0100 split_info:SplitMergeInfo
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
```

```
  aborted:Bool destroyed:Bool
  = TransactionDescr;
trans_split_install$0101 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  installed:Bool = TransactionDescr;
```

Notice that the split install transaction for the new sibling account ξ' refers to its corresponding split prepare transaction of the previously existing account ξ.

4.3.14. Merge prepare and install transactions. A merge prepare transaction converts the state and balance of an account into a message, and a subsequent merge install transaction processes this state:

```
trans_merge_prepare$0110 split_info:SplitMergeInfo
  storage_ph:TrStoragePhase aborted:Bool
  = TransactionDescr;
trans_merge_install$0111 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool
  = TransactionDescr;
```

4.4 Invoking smart contracts in TVM

This section describes the exact parameters with which TVM is invoked during the computing phase of ordinary and other transactions.

4.4.1. Smart-contract code. The *code* of a smart contract is normally a part of the account's persistent state, at least if the account is *active* (cf. **4.1.6**). However, a frozen or uninitialized (or non-existent) account has no persistent state, with the possible exception of the account's balance and the hash of its intended state (equal to the account address for uninitialized accounts). In this case, the code must be supplied in the `init` field of the inbound message being processed by the transaction (cf. **3.1.7**).

4.4.2. Smart-contract persistent data. The *persistent data* of a smart contract is kept alongside its code, and remarks similar to those made above in **4.4.1** apply. In this respect, the code and persistent data of a smart

contract are just two parts of its persistent state, which differ only in the way they are treated by TVM during smart-contract execution.

4.4.3. Smart-contract library environment. The *library environment* of a smart contract is a hashmap mapping 256-bit cell (representation) hashes into the corresponding cells themselves. When an external cell reference is accessed during the execution of a smart contract, the cell referred to is looked up in the library environment and the external cell reference is transparently replaced by the cell found.

The library environment for an invocation of a smart contract is computed as follows:

1. The global library environment for the workchain in question is taken from the current state of the masterchain.[34]

2. Next, it is augmented by the local library environment of the smart contract, stored in the `library` field of the smart contract's state. Only 256-bit keys equal to the hashes of the corresponding value cells are taken into account. If a key is present in both the global and local library environments, the local environment takes precedence while merging the two library environments.

3. Finally, the message library stored in the `library` field of the `init` field of the inbound message is similarly taken into account. Notice, however, that if the account is frozen or uninitialized, the `library` field of the message is part of the suggested state of the account, and is used instead of the local library environment in the previous step. The message library has lower precedence than both the local and the global library environments.

4.4.4. The initial state of TVM. A new instance of TVM is initialized prior to the execution of a smart contract as follows:

- The original `cc` (current continuation) is initialized using the cell slice created from the cell `code`, containing the code of the smart contract computed as described in **4.4.1**.

[34]The most common way of creating shared libraries for TVM is to publish a reference to the root cell of the library in the masterchain.

- The `cp` (TVM codepage) is set to zero. If the smart contract wants to use another TVM codepage x, it must switch to it by using `SETCODEPAGE` x as the first instruction of its code.
- Control register `c0` (return continuation) is initialized by extraordinary continuation `ec_quit` with parameter 0. When executed, this continuation leads to a termination of TVM with exit code 0.
- Control register `c1` (alternative return continuation) is initialized by extraordinary continuation `ec_quit` with parameter 1. When invoked, it leads to a termination of TVM with exit code 1. (Notice that terminating with exit code 0 or 1 is considered a successful termination.)
- Control register `c2` (exception handler) is initialized by extraordinary continuation `ec_quit_exc`. When invoked, it takes the top integer from the stack (equal to the exception number) and terminates TVM with exit code equal to that integer. In this way, by default all exceptions terminate the smart-contract execution with exit code equal to the exception number.
- Control register `c3` (code dictionary) is initialized by the cell with the smart-contract code, similarly to the initial current continuation (`cc`).
- Control register `c4` (root of persistent data) is initialized by the persistent data of the smart contract.[35]
- Control register `c5` (root of actions) is initialized by an empty cell. The "output action" primitives of TVM, such as `SENDMSG`, use `c5` to accumulate the list of actions (e.g., outbound messages) to be performed upon successful termination of the smart contract (cf. **4.2.7** and **4.2.8**).
- Control register `c7` (root of temporary data) is initialized by a singleton *Tuple*, the only component of which is a *Tuple* containing an instance of *SmartContractInfo* with smart contract balance and other useful information (cf. **4.4.10**). The smart contract may replace the temporary data, especially all components of the *Tuple* at `c7` but the first one, with whatever other temporary data it may require. However,

[35]The persistent data of the smart contract need not be loaded in its entirety for this to occur. Instead the root is loaded, and TVM may load other cells by their references from the root only when they are accessed, thus providing a form of virtual memory.

the original content of the *SmartContractInfo* at the first component of the *Tuple* held in `c7` is inspected and sometimes modified by `SENDMSG` TVM primitives and other "output action" primitives of TVM.

- The *gas limits* `gas` $= (g_m, g_l, g_c, g_r)$ are initialized as follows:
  - The *maximal gas limit* g_m is set to the lesser of either the total Gram balance of the smart contract (after the the credit phase—i.e., combined with the value of the inbound message) divided by the current gas price, or the per-execution global gas limit.[36]
  - The *current gas limit* g_l is set to the lesser of either the Gram value of the inbound message divided by the gas price, or the global per-execution gas limit. In this way, always $g_l \leq g_m$. For inbound external messages $g_l = 0$, since they cannot carry any value.
  - The *gas credit* g_c is set to zero for inbound internal messages, and to the lesser of either g_m or a fixed small value (the default external message gas credit, a configurable parameter) for inbound external messages.
  - Finally, the *remaining gas limit* g_r is automatically initialized by $g_l + g_c$.

4.4.5. The initial stack of TVM for processing an internal message. After TVM is initialized as described in **4.4.4**, its stack is initialized by pushing the arguments to the `main()` function of the smart contract as follows:

- The Gram balance b of the smart contract (after crediting the value of the inbound message) is passed as an *Integer* amount of nanograms.
- The Gram balance b_m of inbound message m is passed as an *Integer* amount of nanograms.
- The inbound message m is passed as a cell, which contains a serialized value of type *Message* X, where X is the type of the message body.

[36] Both the global gas limit and the gas price are configurable parameters determined by the current state of the masterchain.

- The body $m_b : X$ of the inbound message, equal to the value of field `body` of m, is passed as a cell slice.
- Finally, the *function selector* s, an *Integer* normally equal to zero, is pushed into the stack.

After that, the code of the smart contract, equal to its initial value of `c3`, is executed. It selects the correct function according to s, which is expected to process the remaining arguments to the function and terminate afterwards.

4.4.6. Processing an inbound external message. An inbound external message is processed similarly to **4.4.4** and **4.4.5**, with the following modifications:

- The function selector s is set to -1, not to 0.
- The Gram balance b_m of inbound message is always 0.
- The initial current gas limit g_l is always 0. However, the initial gas credit $g_c > 0$.

The smart contract must terminate with $g_c = 0$ or $g_r \geq g_c$; otherwise, the transaction and the block containing it are invalid. Validators or collators suggesting a block candidate must never include transactions processing inbound external messages that are invalid.

4.4.7. Processing tick and tock transactions. The TVM stack for processing tick and tock transactions (cf. **4.2.4**) is initialized by pushing the following values:

- The Gram balance b of the current account in nanograms (an *Integer*).
- The 256-bit address ξ of the current account inside the masterchain, represented by an unsigned *Integer*.
- An integer equal to 0 for tick transactions and to -1 for tock transactions.
- The function selector s, equal to -2.

4.4.8. Processing split prepare transactions. For processing split prepare transactions (cf. **4.3.13**), the TVM stack is initialized by pushing the following values:

- The Gram balance b of the current account.
- A *Slice* containing *SplitMergeInfo* (cf. **4.3.13**).
- The 256-bit address ξ of the current account.
- The 256-bit address $\tilde{\xi}$ of the sibling account.
- An integer $0 \leq d \leq 63$, equal to the position of the only bit in which ξ and $\tilde{\xi}$ differ.
- The function selector s, equal to -3.

4.4.9. Processing merge install transactions. For processing merge install transactions (cf. **4.3.14**), the TVM stack is initialized by pushing the following values:

- The Gram balance b of the current account (already combined with the Gram balance of the sibling account).
- The Gram balance b' of the sibling account, taken from the inbound message m.
- The message m from the sibling account, automatically generated by a merge prepare transaction. Its `init` field contains the final state $\tilde{S}$ of the sibling account.
- The state $\tilde{S}$ of the sibling account, represented by a *StateInit* (cf. **3.1.7**).
- A *Slice* containing *SplitMergeInfo* (cf. **4.3.13**).
- The 256-bit address ξ of the current account.
- The 256-bit address $\tilde{\xi}$ of the sibling account.
- An integer $0 \leq d \leq 63$, equal to the position of the only bit in which ξ and $\tilde{\xi}$ differ.
- The function selector s, equal to -4.

4.4.10. Smart-contract information. The smart-contract information structure *SmartContractInfo*, passed in the first component of the *Tuple* contained in control register `c7`, is also a *Tuple* containing the following data:

```
[ magic:0x076ef1ea actions:Integer msgs_sent:Integer
  unixtime:Integer block_lt:Integer trans_lt:Integer
  rand_seed:Integer balance_remaining:[Integer (Maybe Cell)]
  myself:MsgAddressInt global_config:(Maybe Cell)
] = SmartContractInfo;
```

In other words, the first component of this tuple is an *Integer* `magic` always equal to `0x076ef1ea`, the second component is an *Integer* `actions`, originally initialized by zero, but incremented by one whenever an output action is installed by a non-`RAW` output action primitive of the TVM, and so on. The remaining balance is represented by a pair, i.e., a two-component *Tuple*: the first component is the nanogram balance, and the second component is a dictionary with 32-bit keys representing all other currencies, if any (cf. **3.1.6**).

The `rand_seed` field (an unsigned 256-bit integer) here is initialized deterministically starting from the `rand_seed` of the block, the account address, the hash of the inbound message being processed (if any), and the transaction logical time `trans_lt`.

4.4.11. Serialization of output actions. The *output actions* of a smart contract are accumulated in a linked list stored in control register `c5`. The list of output actions is serialized as a value of type *OutList* n, where n is the length of the list:

```
out_list_empty$_ = OutList 0;
out_list$_ {n:#} prev:^(OutList n) action:OutAction
  = OutList (n + 1);
action_send_msg#0ec3c86d out_msg:^(Message Any) = OutAction;
action_set_code#ad4de08e new_code:^Cell = OutAction;
```

5 Block layout

This chapter presents the block layout used by the TON Blockchain, combining the data structures described separately in previous chapters to produce a complete description of a shardchain block. In addition to the TL-B schemes that define the representation of a shardchain block by a tree of cells, this chapter describes exact serialization formats for the resulting bags (collections) of cells, which are necessary to represent a shardchain block as a file.

Masterchain blocks are similar to shardchain blocks, but have some additional fields. The necessary modifications are discussed separately in **5.2**.

5.1 Shardchain block layout

This section lists the data structures that must be contained in a shardchain block and in the shardchain state, and concludes by presenting a formal TL-B scheme for a shardchain block.

5.1.1. Components of the shardchain state. The shardchain state consists of:

- *ShardAccounts*, the split part of the shardchain state (cf. **1.2.2**) containing the state of all accounts assigned to this shard (cf. **4.1.9**).
- *OutMsgQueue*, the output message queue of the shardchain (cf. **3.3.6**).
- *SharedLibraries*, the description of all shared libraries of the shardchain (for now, non-empty only in the masterchain).
- The logical time and the unixtime of the last modification of the state.
- The total balance of the shard.
- A hash reference to the most recent masterchain block, indirectly describing the state of the masterchain and, through it, the state of all other shardchains of the TON Blockchain (cf. **1.5.2**).

5.1.2. Components of a shardchain block. A shardchain block must contain:

- A list of *validator signatures* (cf. **1.2.6**), which is external with respect to all other contents of the block.

- *BlockHeader*, containing general information about the block (cf. **1.2.5**)
- Hash references to the immediately preceding block or blocks of the same shardchain, and to the most recent masterchain block.
- *InMsgDescr* and *OutMsgDescr*, the inbound and outbound message descriptors (cf. **3.2.8** and **3.3.5**).
- *ShardAccountBlocks*, the collection of all transactions processed in the block (cf. **4.2.17**) along with all updates of the states of the accounts assigned to the shard. This is the *split* part of the shardchain block (cf. **1.2.2**).
- The *value flow*, describing the total value imported from the preceding blocks of the same shardchain and from inbound messages, the total value exported by outbound message, the total fees collected by validators, and the total value remaining in the shard.
- A *Merkle update* (cf. [4, 3.1]) of the shardchain state. Such a Merkle update contains the hashes of the initial and final shardchain states with respect to the block, along with all new cells of the final state that have been created while processing the block.[37]

5.1.3. Common parts of the block layout for all workchains. Recall that different workchains may define their own rules for processing messages, other types of transactions, other components of the state, and other ways to serialize all this data. However, some components of the block and its state must be common for all workchains in order to maintain the interoperability between different workchains. Such common components include:

- *OutMsgQueue*, the outbound message queue of a shardchain, which is scanned by neighboring shardchains for messages addressed to them.
- The outer structure of *InMsgDescr* as a hashmap with 256-bit keys equal to the hashes of the imported messages. (The inbound message descriptors themselves need not have the same structure.)

[37] In principle, an experimental version of TON Blockchain might choose to keep only the hashes of the initial and final states of the shardchain. The Merkle update increases the block size, but it is handy for full nodes that want to keep and update their copy of the shardchain state. Otherwise, the full nodes would have to repeat all the computations contained in a block to compute the updated state of the shardchain by themselves.

- Some fields in the block header identifying the shardchain and the block, along with the paths from the block header to the other information indicated in this list.
- The value flow information.

5.1.4. TL-B scheme for the shardchain state. The shardchain state (cf. **1.2.1** and **5.1.1**) is serialized according to the following TL-B scheme:

```
ext_blk_ref$_ start_lt:uint64 end_lt:uint64
  seq_no:uint32 hash:uint256 = ExtBlkRef;

master_info$_ master:ExtBlkRef = BlkMasterInfo;

shard_ident$00 shard_pfx_bits:(## 6)
  workchain_id:int32 shard_prefix:uint64 = ShardIdent;

shard_state shard_id:ShardIdent
  out_msg_queue:OutMsgQueue
  total_balance:CurrencyCollection
  total_validator_fees:CurrencyCollection
  accounts:ShardAccounts
  libraries:(HashmapE 256 LibDescr)
  master_ref:(Maybe BlkMasterInfo)
  custom:(Maybe ^McStateExtra)
  = ShardState;
```

The field `custom` is usually present only in the masterchain and contains all the masterchain-specific data. However, other workchains may use the same cell reference to refer to their specific state data.

5.1.5. Shared libraries description. Shared libraries currently can be present only in masterchain blocks. They are described by an instance of $HashmapE(256, LibDescr)$, where the 256-bit key is the representation hash of the library, and *LibDescr* describes one library:

```
shared_lib_descr$00 lib:^Cell publishers:(Hashmap 256 True)
  = LibDescr;
```

Here `publishers` is a hashmap with keys equal to the addresses of all accounts that have published the corresponding shared library. The shared

library is preserved as long as at least one account keeps it in its published libraries collection.

5.1.6. TL-B scheme for an unsigned shardchain block. The precise format of an *unsigned* (cf. **1.2.6**) shardchain block is given by the following TL-B scheme:

```
block_info version:uint32
  not_master:(## 1)
  after_merge:(## 1) before_split:(## 1) flags:(## 13)
  seq_no:# vert_seq_no:#
  shard:ShardIdent gen_utime:uint32
  start_lt:uint64 end_lt:uint64
  master_ref:not_master?^BlkMasterInfo
  prev_ref:seq_no?^(BlkPrevInfo after_merge)
  prev_vert_ref:vert_seq_no?^(BlkPrevInfo 0)
  = BlockInfo;

prev_blk_info#_ {merged:#} prev:ExtBlkRef
  prev_alt:merged?ExtBlkRef = BlkPrevInfo merged;

unsigned_block info:^BlockInfo value_flow:^ValueFlow
  state_update:^(MERKLE_UPDATE ShardState)
  extra:^BlockExtra = Block;

block_extra in_msg_descr:^InMsgDescr
  out_msg_descr:^OutMsgDescr
  account_blocks:ShardAccountBlocks
  rand_seed:uint256
  custom:(Maybe ^McBlockExtra) = BlockExtra;
```

The field `custom` is usually present only in the masterchain and contains all the masterchain-specific data. However, other workchains may use the same cell reference to refer to their specific block data.

5.1.7. Description of total value flow through a block. The total value flow through a block is serialized according to the following TL-B scheme:

```
value_flow _:^[ from_prev_blk:CurrencyCollection
  to_next_blk:CurrencyCollection
```

```
  imported:CurrencyCollection
  exported:CurrencyCollection ]
  fees_collected:CurrencyCollection
  _:^[
  fees_imported:CurrencyCollection
  created:CurrencyCollection
  minted:CurrencyCollection
  ] = ValueFlow;
```

Recall that `_:^[...]` is a TL-B construction indicating that a group of fields has been moved into a separate cell. The last three fields may be non-zero only in masterchain blocks.

5.1.8. Signed shardchain block. A signed shardchain block is just an unsigned block augmented by a collection of validator signatures:

```
ed25519_signature#5 R:uint256 s:uint256 = CryptoSignature;

signed_block block:^Block blk_serialize_hash:uint256
  signatures:(HashmapE 64 CryptoSignature)
  = SignedBlock;
```

The *serialization hash* `blk_serialize_hash` of the unsigned block `block` is essentially a hash of a specific serialization of the block into an octet string (cf. **5.3.12** for a more detailed explanation). The signatures collected in `signatures` are Ed25519-signatures (cf. **A.3**) made with a validator's private keys of the SHA256 of the concatenation of the 256-bit representation hash of the block `block` and of its 256-bit serialization hash `blk_serialize_hash`. The 64-bit keys in dictionary `signatures` represent the first 64 bits of the public keys of the corresponding validators.

5.1.9. Serialization of a signed block. The overall procedure of serializing and signing a block may be described as follows:

1. An unsigned block B is generated, transformed into a complete bag of cells (cf. **5.3.2**), and serialized into an octet string S_B.

2. Validators sign the 256-bit combined hash

$$H_B := \text{SHA256}\big(\text{HASH}_\infty(B).\,\text{HASH}_M(S_B)\big) \tag{18}$$

   of the representation hash of B and of the Merkle hash of its serialization S_B.

3. A signed shardchain block $\tilde{B}$ is generated from B and these validator signatures as described above (cf. **5.1.8**).

4. This signed block $\tilde{B}$ is transformed into an incomplete bag of cells, which contains only the validator signatures, but the unsigned block itself is absent from this bag of cells, being its only absent cell.

5. This incomplete bag of cells is serialized, and its serialization is prepended to the previously constructed serialization of the unsigned block.

The result is the serialization of the signed block into an octet string. It may be propagated by network or stored into a disk file.

5.2 Masterchain block layout

Masterchain blocks are very similar to shardchain blocks of the basic workchain. This section lists some of the modifications needed to obtain the description of a masterchain block from the description of a shardchain block given in **5.1**.

5.2.1. Additional components present in the masterchain state. In addition to the components listed in **5.1.1**, the masterchain state must contain:

- *ShardHashes* — Describes the current shard configuration, and contains the hashes of the latest blocks of the corresponding shardchains.
- *ShardFees* — Describes the total fees collected by the validators of each shardchain.
- *ShardSplitMerge* — Describes future shard split/merge events. It is serialized as a part of *ShardHashes*.
- *ConfigParams* — Describes the values of all configurable parameters of the TON Blockchain.

5.2.2. Additional components present in masterchain blocks. In addition to the components listed in **5.1.2**, each masterchain block must contain:

- *ShardHashes* — Describes the current shard configuration, and contains the hashes of the latest blocks of the corresponding shardchains. (Notice that this component is also present in the masterchain state.)

5.2.3. Description of *ShardHashes*. *ShardHashes* is represented by a dictionary with 32-bit *workchain_id*s as keys, and "shard binary trees", represented by TL-B type *BinTree ShardDescr*, as values. Each leaf of this shard binary tree contains a value of type *ShardDescr*, which describes a single shard by indicating the sequence number `seq_no`, the logical time `lt`, and the hash `hash` of the latest (signed) block of the corresponding shardchain.

```
bt_leaf$0 {X:Type} leaf:X = BinTree X;
bt_fork$1 {X:Type} left:^(BinTree X) right:^(BinTree X)
          = BinTree X;

fsm_none$0 = FutureSplitMerge;
fsm_split$10 mc_seqno:uint32 = FutureSplitMerge;
fsm_merge$11 mc_seqno:uint32 = FutureSplitMerge;

shard_descr$_ seq_no:uint32 lt:uint64 hash:uint256
  split_merge_at:FutureSplitMerge = ShardDescr;

_ (HashmapE 32 ^(BinTree ShardDescr)) = ShardHashes;
```

Fields `mc_seqno` of `fsm_split` and `fsm_merge` are used to signal future shard merge or split events. Shardchain blocks referring to masterchain blocks with sequence numbers up to, but not including, the one indicated in `mc_seqno` are generated in the usual way. Once the indicated sequence number is reached, a shard merge or split event must occur.

Notice that the masterchain itself is omitted from *ShardHashes* (i.e., 32-bit index -1 is absent from this dictionary).

5.2.4. Description of *ShardFees*. *ShardFees* is a masterchain structure used to reflect the total fees collected so far by the validators of a shardchain. The total fees reflected in this structure are accumulated in the masterchain by crediting them to a special account, whose address is a configurable parameter. Typically this account is the smart contract that computes and distributes the rewards to all validators.

```
bta_leaf$0 {X:Type} {Y:Type} leaf:X extra:Y = BinTreeAug X Y;
```

```
bta_fork$1 {X:Type} {Y:Type} left:^(BinTreeAug X Y)
           right:^(BinTreeAug X Y) extra:Y = BinTreeAug X Y;

_ (HashmapAugE 32 ^(BinTreeAug True CurrencyCollection)
  CurrencyCollection) = ShardFees;
```

The structure of *ShardFees* is similar to that of *ShardHashes* (cf. **5.2.3**), but the dictionary and binary trees involved are augmented by currency values, equal to the `total_validator_fees` values of the final states of the corresponding shardchain blocks. The value aggregated at the root of *ShardFees* is added together with the `total_validator_fees` of the masterchain state, yielding the total TON Blockchain validator fees. The increase of the value aggregated at the root of *ShardFees* from the initial to the final state of a masterchain block is reflected in the `fees_imported` in the value flow of that masterchain block.

5.2.5. Description of *ConfigParams*. Recall that the *configurable parameters* or the *configuration dictionary* is a dictionary `config` with 32-bit keys kept inside the first cell reference of the persistent data of the configuration smart contract γ (cf. **1.6**). The address γ of the configuration smart contract and a copy of the configuration dictionary are duplicated in fields `config_addr` and `config` of a *ConfigParams* structure, explicitly included into masterchain state to facilitate access to the current values of the configurable parameters (cf. **1.6.3**):

```
_ config_addr:uint256 config:^(Hashmap 32 ^Cell)
  = ConfigParams;
```

5.2.6. Masterchain state data. The data specific to the masterchain state is collected into *McStateExtra*, already mentioned in **5.1.4**:

```
masterchain_state_extra#cc1f
  shard_hashes:ShardHashes
  shard_fees:ShardFees
  config:ConfigParams
= McStateExtra;
```

5.2.7. Masterchain block data. Similarly, the data specific to the masterchain blocks is collected into *McBlockExtra*:

```
masterchain_block_extra#cc9f
  shard_hashes:ShardHashes
= McBlockExtra;
```

5.3 Serialization of a bag of cells

The description provided in the previous section defines the way a shardchain block is represented as a tree of cells. However, this tree of cells needs to be serialized into a file, suitable for disk storage or network transfer. This section discusses the standard ways of serializing a tree, a DAG, or a bag of cells into an octet string.

5.3.1. Transforming a tree of cells into a bag of cells. Recall that values of arbitrary (dependent) algebraic data types are represented in the TON Blockchain by *trees of cells*. Such a tree of cells is transformed into a directed acyclic graph, or *DAG*, of cells, by identifying identical cells in the tree. After that, we might replace each of the references of each cell by the 32-byte representation hash of the cell referred to and obtain a *bag of cells*. By convention, the root of the original tree of cells is a marked element of the resulting bag of cells, so that anybody receiving this bag of cells and knowing the marked element can reconstruct the original DAG of cells, hence also the original tree of cells.

5.3.2. Complete bags of cells. Let us say that a bag of cells is *complete* if it contains all cells referred to by any of its cells. In other words, a complete bag of cells does not have any "unresolved" hash references to cells outside that bag of cells. In most cases, we need to serialize only complete bags of cells.

5.3.3. Internal references inside a bag of cells. Let us say that a reference of a cell c belonging to a bag of cells B is *internal (with respect to B)* if the cell c_i referred to by this reference belongs to B as well. Otherwise, the reference is called *external*. A bag of cells is complete if and only if all references of its constituent cells are internal.

5.3.4. Assigning indices to the cells from a bag of cells. Let $c_0, \ldots, c_{n-1}$ be the n distinct cells belonging to a bag of cells B. We can list these cells in some order, and then assign indices from 0 to $n-1$, so that cell c_i gets index i. Some options for ordering cells are:

- Order cells by their representation hash. Then $\text{Hash}(c_i) < \text{Hash}(c_j)$ whenever $i < j$.
- Topological order: if cell c_i refers to cell c_j, then $i < j$. In general, there is more than one topological order for the same bag of cells. There are two standard ways for constructing topological orders:
  - Depth-first order: apply a depth-first search to the directed acyclic graph of cells starting from its root (i.e., marked cell), and list cells in the order they are visited.
  - Breadth-first order: same as above, but applying a breadth-first search.

Notice that the topological order always assigns index 0 to the root cell of a bag of cells constructed from a tree of cells. In most cases, we opt to use a topological order, or the depth-first order if we want to be more specific.

If cells are listed in a topological order, then the verification that there are no cyclic references in a bag of cells is immediate. On the other hand, ordering cells by their representation hash simplifies the verification that there are no duplicates in a serialized bag of cells.

5.3.5. Outline of serialization process. The serialization process of a bag of cells B consisting of n cells can be outlined as follows:

1. List the cells from B in a topological order: c_0, c_1, ..., c_{n-1}. Then c_0 is the root cell of B.
2. Choose an integer s, such that $n \leq 2^s$. Represent each cell c_i by an integral number of octets in the standard way (cf. **1.1.3** or [4, 3.1.4]), but using unsigned big-endian s-bit integer j instead of hash $\text{Hash}(c_j)$ to represent internal references to cell c_j (cf. **5.3.6** below).
3. Concatenate the representations of cells c_i thus obtained in the increasing order of i.
4. Optionally, an index can be constructed that consists of $n + 1$ t-bit integer entries L_0, ..., L_n, where L_i is the total length (in octets) of the representations of cells c_j with $j \leq i$, and integer $t \geq 0$ is chosen so that $L_n \leq 2^t$.

5. The serialization of the bag of cells now consists of a magic number indicating the precise format of the serialization, followed by integers $s \geq 0$, $t \geq 0$, $n \leq 2^s$, an optional index consisting of $\lceil (n+1)t/8 \rceil$ octets, and L_n octets with the cell representations.

6. An optional CRC32 may be appended to the serialization for integrity verification purposes.

If an index is included, any cell c_i in the serialized bag of cells may be easily accessed by its index i without deserializing all other cells, or even without loading the entire serialized bag of cells in memory.

5.3.6. Serialization of one cell from a bag of cells. More precisely, each individual cell $c = c_i$ is serialized as follows, provided s is a multiple of eight (usually $s = 8$, 16, 24, or 32):

1. Two descriptor bytes d_1 and d_2 are computed similarly to [4, 3.1.4] by setting $d_1 = r + 8s + 16h + 32l$ and $d_2 = \lfloor b/8 \rfloor + \lceil b/8 \rceil$, where:

   - $0 \leq r \leq 4$ is the number of cell references present in cell c; if c is absent from the bag of cells being serialized and is represented by its hashes only, then $r = 7$.[38]
   - $0 \leq b \leq 1023$ is the number of data bits in cell c.
   - $0 \leq l \leq 3$ is the level of cell c (cf. [4, 3.1.3]).
   - $s = 1$ for exotic cells and $s = 0$ for ordinary cells.
   - $h = 1$ if the cell's hashes are explicitly included into the serialization; otherwise, $h = 0$. (When $r = 7$, we must always have $h = 1$.)

   For absent cells (i.e., external references), only d_1 is present, always equal to $23 + 32l$.

2. Two bytes d_1 and d_2 (if $r < 7$) or one byte d_1 (if $r = 7$) begin the serialization of cell c.

[38] Notice that these "absent cells" are different from the library reference and external reference cells, which are kinds of exotic cells (cf. [4, 3.1.7]). Absent cells, by contrast, are introduced only for the purpose of serializing incomplete bags of cells, and can never be processed by TVM.

3. If $h = 1$, the serialization is continued by $l + 1$ 32-byte higher hashes of c (cf. [4, 3.1.6]): $\text{Hash}_1(c)$, ..., $\text{Hash}_{l+1}(c) = \text{Hash}_\infty(c)$.

4. After that, $\lceil b/8 \rceil$ data bytes are serialized, by splitting b data bits into 8-bit groups and interpreting each group as a big-endian integer in the range $0 \ldots 255$. If b is not divisible by 8, then the data bits are first augmented by one binary `1` and up to six binary `0`, so as to make the number of data bits divisible by eight.[39]

5. Finally, r cell references to cells c_{j_1}, ..., c_{j_r} are encoded by means of r s-bit big-endian integers j_1, ..., j_r.[40]

5.3.7. A classification of serialization schemes for bags of cells. A serialization scheme for a bag of cells must specify the following parameters:

- The 4-byte magic number prepended to the serialization.
- The number of bits s used to represent cell indices. Usually s is a multiple of eight (e.g., 8, 16, 24, or 32).
- The number of bits t used to represent offsets of cell serializations (cf. **5.3.5**). Usually t is also a multiple of eight.
- A flag indicating whether an index with offsets L_0, ..., L_n of cell serializations is present. This flag may be combined with t by setting $t = 0$ when the index is absent.
- A flag indicating whether the CRC32-C of the whole serialization is appended to it for integrity verification purposes.

5.3.8. Fields present in the serialization of a bag of cells. In addition to the values listed in **5.3.7**, fixed by the choice of a serialization scheme for bags of cells, the serialization of a specific bag of cells must specify the following parameters:

- The total number of cells n present in the serialization.

[39] Notice that exotic cells (with $s = 1$) always have $b \geq 8$, with the cell type encoded in the first eight data bits (cf. [4, 3.1.7]).

[40] If the bag of cells is not complete, some of these cell references may refer to cells c' absent from the bag of cells. In that case, special "absent cells" with $r = 7$ are included into the bag of cells and are assigned some indices j. These indices are then used to represent references to absent cells.

- The number of "root cells" $k \leq n$ present in the serialization. The root cells themselves are c_0, ..., c_{k-1}. All other cells present in the bag of cells are expected to be reachable by chains of references starting from the root cells.
- The number of "absent cells" $l \leq n - k$, which represent cells that are actually absent from this bag of cells, but are referred to from it. The absent cells themselves are represented by c_{n-l}, ..., c_{n-1}, and only these cells may (and also must) have $r = 7$. Complete bags of cells have $l = 0$.
- The total length in bytes L_n of the serialization of all cells. If the index is present, L_n might not be stored explicitly since it can be recovered as the last entry of the index.

5.3.9. TL-B scheme for serializing bags of cells. Several TL-B constructors can be used to serialize bags of cells into octet (i.e., 8-bit byte) sequences. The only one that is currently used to serialize new bags of cell is

```
serialized_boc#b5ee9c72 has_idx:(## 1) has_crc32c:(## 1)
  has_cache_bits:(## 1) flags:(## 2) { flags = 0 }
  size:(## 3) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots >= 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  root_list:(roots * ##(size * 8))
  index:has_idx?(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:has_crc32c?uint32
  = BagOfCells;
```

Field `cells` is n, `roots` is k, `absent` is l, and `tot_cells_size` is L_n (the total size of the serialization of all cells in bytes). If an index is present, parameters $s/8$ and $t/8$ are serialized separately as `size` and `off_bytes`, respectively, and the flag `has_idx` is set. The index itself is contained in `index`, present only if `has_idx` is set. The field `root_list` contains the (zero-based) indices of the root nodes of the bag of cells.

Two older constructors are still supported in the bag-of-cells deserialization functions:

```
serialized_boc_idx#68ff65f3 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  = BagOfCells;

serialized_boc_idx_crc32c#acc3a728 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:uint32 = BagOfCells;
```

5.3.10. Storing compiled TVM code in files. Notice that the above procedure for serializing bags of cells may be used to serialize compiled smart contracts and other TVM code. One must define a TL-B constructor similar to the following:

```
compiled_smart_contract
  compiled_at:uint32 code:^Cell data:^Cell
  description:(Maybe ^TinyString)
  _:^[ source_file:(Maybe ^TinyString)
       compiler_version:(Maybe ^TinyString) ]
  = CompiledSmartContract;

tiny_string#_ len:(#<= 126) str:(len * [ uint8 ]) = TinyString;
```

Then a compiled smart contract may be represented by a value of type *CompiledSmartContract*, transformed into a tree of cells and then into a bag of

cells, and then serialized using one of the constructors listed in **5.3.9**. The resulting octet string may be then written into a file with suffix `.tvc` ("TVM smart contract"), and this file may be used to distribute the compiled smart contract, download it into a wallet application for deploying into the TON Blockchain, and so on.

5.3.11. Merkle hashes for an octet string. On some occasions, we must define a Merkle hash $\textsc{Hash}_M(s)$ of an arbitrary octet string s of length $|s|$. We do this as follows:

- If $|s| \leq 256$ octets, then the Merkle hash of s is just its $\textsc{sha256}$:

$$\textsc{Hash}_M(s) := \textsc{sha256}(s) \quad \text{if } |s| \leq 256. \tag{19}$$

- If $|s| > 256$, let $n = 2^k$ be the largest power of two less than $|s|$ (i.e., $k := \lfloor \log_2(|s|-1) \rfloor$, $n := 2^k$). If s' is the prefix of s of length n, and s'' is the suffix of s of length $|s| - n$, so that s is the concatenation $s'.s''$ of s' and s'', we define

$$\textsc{Hash}_M(s) := \textsc{sha256}\big(\textsc{int}_{64}(|s|).\,\textsc{Hash}_M(s').\,\textsc{Hash}_M(s'')\big) \tag{20}$$

  In other words, we concatenate the 64-bit big-endian representation of $|s|$ and the recursively computed Merkle hashes of s' and s'', and compute $\textsc{sha256}$ of the resulting string.

One can check that $\textsc{Hash}_M(s) = \textsc{Hash}_M(t)$ for octet strings s and t of length less than $2^{64} - 2^{56}$ implies $s = t$ unless a hash collision for $\textsc{sha256}$ has been found.

5.3.12. The serialization hash of a block. The construction of **5.3.11** is applied in particular to the serialization of the bag of cells representing an unsigned shardchain or masterchain block. The validators sign not only the representation hash of the unsigned block, but also the "serialization hash" of the unsigned block, defined as $\textsc{Hash}_M$ of the serialization of the unsigned block. In this way, the validators certify that this octet string is indeed a serialization of the corresponding block.

References

[1] Daniel J. Bernstein, *Curve25519: New Diffie–Hellman Speed Records* (2006), in: M. Yung, Ye. Dodis, A. Kiayas et al, *Public Key Cryptography*, Lecture Notes in Computer Science **3958**, pp. 207–228. Available at `https://cr.yp.to/ecdh/curve25519-20060209.pdf`.

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange et al., *High-speed high-security signatures* (2012), *Journal of Cryptographic Engineering* **2** (2), pp. 77–89. Available at `https://ed25519.cr.yp.to/ed25519-20110926.pdf`.

[3] N. Durov, *Telegram Open Network*, 2017.

[4] N. Durov, *Telegram Open Network Virtual Machine*, 2018.

A Elliptic curve cryptography

This appendix contains a formal description of the elliptic curve cryptography currently used in TON, particularly in the TON Blockchain and the TON Network.

TON uses two forms of elliptic curve cryptography: Ed25519 is used for cryptographic Schnorr signatures, while Curve25519 is used for asymmetric cryptography. These curves are used in the standard way (as defined in the original articles [1] and [2] by D. Bernstein and RFCs 7748 and 8032); however, some serialization details specific to TON must be explained. One unique adaptation of these curves for TON is that TON supports automatic conversion of Ed25519 keys into Curve25519 keys, so that the same keys can be used for signatures and for asymmetric cryptography.

A.1 Elliptic curves

Some general facts on elliptic curves over finite fields, relevant for elliptic curve cryptography, are collected in this section.

A.1.1. Finite fields. We consider elliptic curves over finite fields. For the purposes of the Curve25519 and Ed25519 algorithms, we will be mostly concerned with elliptic curves over the finite prime field $k := \mathbb{F}_p$ of residues modulo p, where $p = 2^{255} - 19$ is a prime number, and over finite extensions $\mathbb{F}_q$ of $\mathbb{F}_p$, especially the quadratic extension $\mathbb{F}_{p^2}$.[41]

A.1.2. Elliptic curves. An *elliptic curve* $E = (E, O)$ over a field k is a geometrically integral smooth projective curve E/k of genus $g = 1$, along with a marked k-rational point $O \in E(k)$. It is well-known that an elliptic curve E over a field k can be represented in (generalized) Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad \text{for some } a_1, \ldots, a_6 \in k. \tag{21}$$

More precisely, this is only the affine part of the elliptic curve, written in coordinates (x, y). For any field extension K of k, $E(K)$ consists of all solutions $(x, y) \in K^2$ of equation (21), called *finite points of* $E(K)$, along with a point at infinity, which is the marked point O.

[41] Arithmetic modulo p for a modulus p near a power of two can be implemented very efficiently. On the other hand, residues modulo $2^{255} - 19$ can be represented by 255-bit integers. This is the reason this particular value of p has been chosen by D. Bernstein.

A.1.3. Weierstrass form in homogeneous coordinates. In homogeneous coordinates $[X : Y : Z]$, (21) corresponds to

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{22}$$

When $Z \neq 0$, we can set $x := X/Z$, $y := Y/Z$, and obtain a solution (x, y) of (21) (i.e., a finite point of E). On the other hand, the only solution (up to proportionality) of (22) with $Z = 0$ is $[0 : 1 : 0]$; this is the point at infinity O.

A.1.4. Standard Weierstrass form. When the characteristic $\operatorname{char} k$ of field k is $\neq 2$, 3, the Weierstrass form of (21) or (22) can be simplified with the aid of linear transformations $y' := y + a_1x/2 + a_3/2$, $x' := x + a_2/3$, thus making $a_1 = a_3 = a_2 = 0$ and obtaining

$$y^2 = x^3 + a_4x + a_6 \tag{23}$$

and

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \tag{24}$$

Such an equation defines an elliptic curve if and only if the cubic polynomial $P(x) := x^3 + a_4x + a_6$ has no multiple roots, i.e., if the discriminant $D := -4a_4^3 - 27a_6^2$ is non-zero.

A.1.5. Addition of points on elliptic curve E. Let K be a field extension of field k, and let $E = (E, O)$ be any elliptic curve in Weierstrass form defined over k. Then any line $l \subset \mathbb{P}^2_K$ intersects the elliptic curve $E_{(K)}$ (which is the base change of curve E to field K, i.e., the curve defined by the same equations over a larger field K) at exactly three points P, Q, R, considered with multiplicities. We define the *addition of points* on elliptic curve E (or rather the addition of its K-valued points $E(K)$) by requiring that

$$P + Q + R = O \quad \text{whenever } \{P, Q, R\} = l \cap E \text{ for some line } l \subset \mathbb{P}^2_K. \tag{25}$$

It is well-known that this requirement defines a unique commutative law $[+] : E \times_k E \to E$ on the points of the elliptic curve E, having O as its neutral element. When elliptic curve E is represented by its Weierstrass form (21), one can write explicit formulas expressing the coordinates x_{P+Q}, y_{P+Q} of the sum $P+Q$ of two K-valued points P, $Q \in E(K)$ of elliptic curve E as rational functions of the coordinates x_P, y_P, x_Q, $y_Q \in K$ of points P and Q and of the coefficients $a_i \in k$ of (21).

A.1.6. Power maps. Since $E(K)$ is an abelian group, one can define *multiples* or *powers* $[n]X$ for any point $X \in E(K)$ and any integer $n \in \mathbb{Z}$. If $n = 0$, then $[0]X = O$; if $n > 0$, then $[n]X = [n-1]X + X$; if $n < 0$, then $[n]X = -[-n]X$. The map $[n] = [n]_E : E \to E$ for $n \neq 0$ is an *isogeny*, meaning that it is a non-constant homomorphism for the group law of E:

$$[n](P+Q) = [n]P + [n]Q \quad \text{for any } P, Q \in E(K) \text{ and } n \in \mathbb{Z}. \tag{26}$$

In particular, $[-1]_E : E \to E$, $P \mapsto -P$, is an involutive automorphism of elliptic curve E. If E is in Weierstrass form, $[-1]_E$ maps $(x, y) \mapsto (x, -y)$, and two points $P, Q \in E(\mathbb{F}_q)$ have equal x-coordinates if and only if $Q = \pm P$.

A.1.7. The order of the group of rational points of E. Let E be an elliptic curve defined over a finite base field k, and let $K = \mathbb{F}_q$ be a finite extension of k. Then $E(\mathbb{F}_q)$ is a finite abelian group. By a well-known result of Hasse, the order n of this group is not too distant from q:

$$n = |E(\mathbb{F}_q)| = q - t + 1 \quad \text{where } t^2 \leq 4q, \text{ i.e., } |t| \leq 2\sqrt{q}. \tag{27}$$

We will be mostly interested in the case $K = k = \mathbb{F}_p$, with $q = p$ a prime number.

A.1.8. Cyclic subgroups of large prime order. Elliptic curve cryptography is usually performed using elliptic curves that admit a (necessarily cyclic) subgroup $C \subset E(\mathbb{F}_q)$ of prime order ℓ. Equivalently, a rational point $G \in E(\mathbb{F}_q)$ of prime order ℓ may be given; then C can be recovered as the cyclic subgroup $\langle G \rangle$ generated by G. In order to verify that a point $G \in E(\mathbb{F}_q)$ generates a cyclic group of prime order ℓ, one can check that $G \neq O$, but $[\ell]G = O$.

By the Legendre theorem, ℓ is necessarily a divisor of the order $n = |E(\mathbb{F}_q)|$ of the finite abelian group $E(\mathbb{F}_q)$:

$$n = |E(\mathbb{F}_q)| = c\ell \quad \text{for some integer } c \geq 1 \tag{28}$$

The integer c is called the *cofactor*; one usually wants the cofactor to be as small as possible, so as to make $\ell = n/c$ as large as possible. Recall that n always has the same order of magnitude as q by (27), so it cannot be changed much by varying E once q is fixed.

A.1.9. Data for elliptic curve cryptography. In order to define specific elliptic curve cryptography, one must fix a finite base field $\mathbb{F}_q$ (if $q = p$ is a prime, it is sufficient to fix prime p), an elliptic curve $E/\mathbb{F}_q$ (usually represented by the coefficients of its Weierstrass form (23) or (21)), the base point O (which usually is the point at infinity of an elliptic curve written in Weierstrass form), and the generator $G \in E(\mathbb{F}_q)$ (usually determined by its coordinates (x, y) with respect to the equation of the elliptic curve) of a cyclic subgroup of large prime order ℓ. Prime number ℓ and the cofactor c are usually also part of the elliptic cryptography data.

A.1.10. Main operations of elliptic curve cryptography. Elliptic curve cryptography usually deals with a fixed cyclic subgroup C of a large prime order ℓ inside the group of points of an elliptic curve E over a finite field $\mathbb{F}_q$. A generator G of C is usually fixed. It is usually assumed that, given a point X of C, one cannot find its "discrete logarithm base G" (i.e., a residue n modulo ℓ such that $X = [n]G$) faster than in $O(\sqrt{\ell})$ operations. The most important operations used in elliptic curve cryptography are the addition of points from $C \subset E(\mathbb{F}_q)$ and the computation of their powers, or multiples.

A.1.11. Private and public keys for elliptic curve cryptography. Usually a private key for elliptic curve cryptography described by the data listed in **A.1.9** is a "random" integer $0 < a < \ell$, called the *secret exponent*, and the corresponding public key is the point $A := [a]G$ (or just its x-coordinate x_A), suitably serialized.

A.1.12. Montgomery curves. Elliptic curves with the specific Weierstrass equation

$$y^2 = x^3 + Ax^2 + x \quad \text{where } A = 4a - 2 \text{ for some } a \in k,\ a \neq 0,\ a \neq 1 \tag{29}$$

are called *Montgomery curves*. They have the convenient property that $x_{P+Q}x_{P-Q}$ can be expressed as a simple rational function of x_P and x_Q:

$$x_{P+Q}x_{P-Q} = \left(\frac{x_Px_Q - A}{x_P - x_Q}\right)^2 \tag{30}$$

This means that x_{P+Q} can be computed provided x_{P-Q}, x_P, and x_Q are known. In particular, if x_P, $x_{[n]P}$, and $x_{[n+1]P}$ are known, then $x_{[2n]P}$, $x_{[2n+1]P}$, and $x_{[2n+2]P}$ can be computed. Using the binary representation of $0 < n < 2^s$, one can compute $x_{[\lfloor n/2^{s-i}\rfloor]P}$, $x_{[\lfloor n/2^{s-i}\rfloor+1]P}$ for $i = 0, 1, \ldots, s$, thus obtaining

$x_{[n]P}$ (this algorithm for quickly computing $x_{[n]P}$ starting from x_P on Montgomery curves is called a *Montgomery ladder*). Hence we see the importance of Montgomery curves for elliptic curve cryptography.

A.2 Curve25519 cryptography

This section describes the well-known Curve25519 cryptography proposed by Daniel Bernstein [1] and its usage in TON.

A.2.1. Curve25519. *Curve25519* is defined as the Montgomery curve

$$y^2 = x^3 + Ax^2 + x \quad \text{over } \mathbb{F}_p, \text{ where } p = 2^{255} - 19 \text{ and } A = 486662. \tag{31}$$

The order of this curve is 8ℓ, where ℓ is a prime number, and $c = 8$ is the cofactor. The cyclic subgroup of order ℓ is generated by a point G with $x_G = 9$ (this determines G up to the sign of y_G, which is unimportant). The order of the quadratic twist $2y^2 = x^3 + Ax^2 + x$ of Curve25519 is $4\ell'$ for another prime number ℓ'.[42]

A.2.2. Parameters of Curve25519. The parameters of Curve25519 are as follows:

- Base field: Prime finite field $\mathbb{F}_p$ for $p = 2^{255} - 19$.
- Equation: $y^2 = x^3 + Ax^2 + x$ for $A = 486662$.
- Base point G: Characterized by $x_G = 9$ (nine is the smallest positive integer x-coordinate of a generator of the subgroup of large prime order of $E(\mathbb{F}_p)$).
- Order of $E(\mathbb{F}_p)$:

$$|E(\mathbb{F}_p)| = p - t + 1 = 8\ell, \quad \text{where} \tag{32}$$

$$\ell = 2^{252} + 27742317777372353535851937790883648493 \quad \text{is prime.} \tag{33}$$

[42] Actually, D. Bernstein chose $A = 486662$ because it is the smallest positive integer $A \equiv 2 \pmod 4$ such that both the corresponding Montgomery curve (31) over $\mathbb{F}_p$ for $p = 2^{255} - 19$ and the quadratic twist of this curve have small cofactors. Such an arrangement avoids the necessity to check whether an x-coordinate $x_P \in \mathbb{F}_p$ of a point P defines a point $(x_P, y_P) \in \mathbb{F}_p^2$ lying on the Montgomery curve itself or on its quadratic twist.

- Order of $\tilde{E}(\mathbb{F}_p)$, where $\tilde{E}$ is the quadratic twist of E:

$$|\tilde{E}(\mathbb{F}_p)| = p + t + 1 = 2p + 2 - 8\ell = 4\ell', \quad \text{where} \tag{34}$$

$$\ell' = 2^{253} - 55484635554744707071703875581767296995 \quad \text{is prime.} \tag{35}$$

A.2.3. Private and public keys for standard Curve25519 cryptography. A private key for Curve25519 cryptography is usually defined as a *secret exponent* a, while the corresponding public key is x_A, the x-coordinate of point $A := [a]G$. This is usually sufficient for performing ECDH (elliptic curve Diffie–Hellman key exchange) and asymmetric elliptic curve cryptography, as follows:

If a party wants to send a message M to another party, which has public key x_A (and private key a), the following computations are performed. A one-time random secret exponent b is generated, and $x_B := x_{[b]G}$ and $x_{[b]A}$ are computed using a Montgomery ladder. After that, the message M is encrypted by a symmetric cypher such as AES using the 256-bit "shared secret" $S := x_{[b]A}$ as a key, and 256-bit integer ("one-time public key") x_B is prepended to the encrypted message. Once the party with public key x_A receives this message, it can compute $x_{[a]B}$ starting from x_B (transmitted with the encrypted message) and the private key a. Since $x_{[a]B} = x_{[ab]G} = x_{[b]A} = S$, the receiving party recovers the shared secret S and is able to decrypt the remainder of the message.

A.2.4. Public and private keys for TON Curve25519 cryptography. TON uses another form for public and private keys of Curve25519 cryptography, borrowed from Ed25519 cryptography.

A private key for TON Curve25519 cryptography is just a random 256-bit string k. It is used by computing $\text{SHA512}(k)$, taking the first 256 bits of the result, interpreting them as a little-endian 256-bit integer a, clearing bits 0, 1, 2, and 255 of a, and setting bit 254 so as to obtain a value $2^{254} \le a < 2^{255}$, divisible by eight. The value a thus obtained is the *secret exponent* corresponding to k; meanwhile, the remaining 256 bits of $\text{SHA512}(k)$ constitute the *secret salt* k''.

The public key corresponding to k—or to the secret exponent a—is just the x-coordinate x_A of the point $A := [a]G$. Once a and x_A are computed, they are used in exactly the same way as in **A.2.3**. In particular, if x_A needs to be serialized, it is serialized into 32 octets as an unsigned little-endian 256-bit integer.

A.2.5. Curve25519 is used in the TON Network. Notice that the asymmetric Curve25519 cryptography described in **A.2.4** is extensively used by the TON Network, especially the ADNL (Abstract Datagram Network Layer) protocol. However, TON Blockchain needs elliptic curve cryptography mostly for signatures. For this purpose, Ed25519 signatures described in the next section are used.

A.3 Ed25519 cryptography

Ed25519 cryptography is extensively used for fast cryptographic signatures by both the TON Blockchain and the TON Network. This section describes the variant of Ed25519 cryptography used by TON. An important difference from the standard approaches (as defined by D. Bernstein et al. in [2]) is that TON provides automatic conversion of private and public Ed25519 keys into Curve25519 keys, so that the same keys could be used both for encrypting/decrypting and for signing messages.

A.3.1. Twisted Edwards curves. A *twisted Edwards curve* $E_{a,d}$ with parameters $a \neq 0$ and $d \neq 0, a$ over a field k is given by equation

$$E_{a,d}: ax^2 + y^2 = 1 + dx^2y^2 \quad \text{over } k \tag{36}$$

If $a = 1$, this equation defines an (untwisted) Edwards curve. Point $O(0, 1)$ is usually chosen as the marked point of $E_{a,d}$.

A.3.2. Twisted Edwards curves are birationally equivalent to Montgomery curves. A twisted Edwards curve $E_{a,d}$ is birationally equivalent to a Montgomery elliptic curve

$$M_A: v^2 = u^3 + Au^2 + u \tag{37}$$

where $A = 2(a + d)/(a - d)$ and $d/a = (A - 2)/(A + 2)$. The birational equivalence $\phi : E_{a,d} \dashrightarrow M_A$ and its inverse ϕ^{-1} are given by

$$\phi : (x, y) \mapsto \left(\frac{1+y}{1-y}, \frac{c(1+y)}{x(1-y)}\right) \tag{38}$$

and

$$\phi^{-1} : (u, v) \mapsto \left(\frac{cu}{v}, \frac{u-1}{u+1}\right) \tag{39}$$

where

$$c = \sqrt{\frac{A+2}{a}} \tag{40}$$

Notice that ϕ transforms the marked point $O(0,1)$ of $E_{a,d}$ into the marked point of M_A (i.e., its point at infinity).

A.3.3. Addition of points on a twisted Edwards curve. Since $E_{a,d}$ is birationally equivalent to an elliptic curve M_A, the addition of points on M_A can be transferred to $E_{a,d}$ by setting

$$P + Q := \phi^{-1}\big(\phi(P) + \phi(Q)\big) \quad \text{for any } P,\, Q \in E_{a,d}(k). \tag{41}$$

Notice that the marked point $O(0,1)$ is the neutral element with respect to this addition, and that $-(x_P, y_P) = (-x_P, y_P)$.

A.3.4. Formulas for adding points on a twisted Edwards curve. The coordinates x_{P+Q} and y_{P+Q} admit simple expressions as rational functions of x_P, y_P, x_Q, y_Q:

$$x_{P+Q} = \frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q} \tag{42}$$

$$y_{P+Q} = \frac{y_P y_Q - a x_P x_Q}{1 - d x_P x_Q y_P y_Q} \tag{43}$$

These expressions can be efficiently computed, especially if $a = -1$. This is the reason twisted Edwards curves are important for fast elliptic curve cryptography.

A.3.5. Ed25519 twisted Edwards curve. Ed25519 is the twisted Edwards curve $E_{-1,d}$ over $\mathbb{F}_p$, where $p = 2^{255} - 19$ is the same prime number as that used for Curve25519, and $d = -(A-2)/(A+2) = -121665/121666$, where $A = 486662$ is the same as in the equation (31):

$$-x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2 \quad \text{for } x,\, y \in \mathbb{F}_p,\, p = 2^{255} - 19. \tag{44}$$

In this way, Ed25519-curve $E_{-1,d}$ is birationally equivalent to Curve25519 (31), and one can use $E_{-1,d}$ and formulas (42)–(43) for point addition on either Ed25519 or Curve25519, using (38) and (39) to convert points on Ed25519 into corresponding points on Curve25519, and vice versa.

A.3.6. Generator of Ed25519. The generator of Ed25519 is the point G' with $y(G') = 4/5$ and $0 \le x(G') < p$ even. According to (38), it corresponds to the point (u, v) of Curve25519 with $u = (1 + 4/5)/(1 - 4/5) = 9$ (i.e., to the generator G of Curve25519 chosen in **A.2.2**). In particular, $G = \phi(G')$, G' generates a cyclic subgroup of the same large prime order ℓ given in (32), and for any integer a,

$$\phi([a]G') = [a]G \quad . \tag{45}$$

In this way, we can perform computations with Curve25519 and its generator G, or with Ed25519 and generator G', and obtain essentially the same results.

A.3.7. Standard representation of points on Ed25519. A point $P(x, y)$ on Ed25519 may be represented by its two coordinates x_P and y_P, residues modulo $p = 2^{255} - 19$. In their turn, both these coordinates may be represented by unsigned 255- or 256-bit integers $0 \le x_P, y_P < p < 2^{255}$.

However, a more compact representation of P by one little-endian unsigned 256-bit integer $\tilde{P}$ is commonly used (and is used by TON as well). Namely, the 255 lower-order bits of $\tilde{P}$ contain y_P, $0 \le y_P < p < 2^{255}$, and bit 255 is used to store $x_P \bmod 2$, the lower-order bit of x_P. Since y_P always determines x_P up to sign (i.e., up to replacing x_P with $p - x_P$), x_P and $p - x_P$ can always be distinguished by their lower-order bit, p being odd.

If it is sufficient to know $\pm P$ up to sign, one can ignore $x_P \bmod 2$ and consider only the little-endian 255-bit integer y_P, setting the bit 255 arbitrarily, ignoring its previously defined value, or clearing it.

A.3.8. Private key for Ed25519. A *private key* for Ed25519 is just an arbitrary 256-bit string k. A *secret exponent* a and *secret salt* k'' are derived from k by first computing $\text{SHA512}(k)$, and then taking the first 256 bits of this SHA512 as the little-endian representation of a (but with bits 255, 2, 1, and 0 cleared, and bit 254 set); the last 256 bits of $\text{SHA512}(k)$ then constitute k''.

This is essentially the same procedure as described in **A.2.4**, but with Curve25519 replaced by the birationally equivalent curve Ed25519. (In fact, it is the other way around: this procedure is standard for Ed25519-based elliptic curve cryptography, and TON extends the procedure to Curve25519.)

A.3.9. Public key for Ed25519. A *public key* corresponding to a private key k for Ed25519 is the standard representation (cf. **A.3.7**) of the point $A = [a]G'$, where a is the secret exponent (cf. **A.3.8**) defined by the private key k.

Notice that $\phi(A)$ is the public key for Curve25519 defined by the same private key k according to **A.2.4** and (45). In this way, we can convert public keys for Ed25519 into corresponding public keys for Curve25519, and vice versa. Private keys do not need to be transformed at all.

A.3.10. Cryptographic Ed25519-signatures. If a message (octet string) M needs to be signed by a private key k defining secret exponent a and secret salt k'', the following computations are performed:

- $r := \text{SHA}512(k''|M)$, interpreted as a little-endian 512-bit integer. Here $s|t$ denotes the concatenation of octet strings s and t.
- $R := [r]G'$ is a point on Ed25519.
- $\tilde{R}$ is the standard representation (cf. **A.3.7**) of point R as a 32-octet string.
- $s := r + a \cdot \text{SHA}512(\tilde{R}|\tilde{A}|M) \bmod \ell$, encoded as a little-endian 256-bit integer. Here $\tilde{A}$ is the standard representation of point $A = [a]G'$, the public key corresponding to k.

The (Schnorr) signature is a 64-octet string $(\tilde{R}, s)$, consisting of the standard representation of the point R and of the 256-bit integer s.

A.3.11. Checking Ed25519-signatures. In order to verify signature $(\tilde{R}, s)$ of a message M, supposedly made by the owner of the private key k corresponding to a known public key A, the following steps are performed:

- Points $[s]G'$ and $R + [\text{SHA}512(\tilde{R}|\tilde{A}|M)]A$ of Ed25519 are computed.
- If these two points coincide, the signature is correct.