

Telegram Open Network Blockchain

Николай Дуров

8 февраля, 2020

Аннотация

Цель данного текста — предоставить детальное описание Telegram Open Network (TON) Blockchain.

Введение

Этот документ предоставляет детальное описание TON Blockchain, включая точный формат блока, условия валидности, детали вызова Виртуальной Машины TON (TVM), процесс создания смарт-контрактов и криптографические подписи. В этом отношении он является продолжением TON whitepaper (см. [3]), поэтому мы свободно используем терминологию, представленную в этом документе.

Глава 1 дает общий обзор TON Blockchain и его принципов проектирования с особым вниманием к введению условий совместимости и валидности, а также реализации гарантий доставки сообщений. Более детальная информация, такая как схемы TL-B, которые описывают сериализацию всех необходимых структур данных в деревья или коллекции («пакеты») ячеек («bags» of cells), предоставляется в последующих главах, кульминируя в полном описании формата блока TON Blockchain (shardchain и masterchain) в Главе 5.

Детальное описание криптографии на эллиптических кривых, используемой для подписи блоков и сообщений, также доступное через примитивы TVM, представлено в Приложении A. Сам TVM описан в отдельном документе (см. [4]).

Некоторые темы намеренно оставлены вне этого документа. Одна из них — это протокол Византийской отказоустойчивости (BFT), используемый валидаторами для определения следующего блока мастерчейна (masterchain) или шардчейна (shardchain); эта тема оставлена для будущего документа, посвященного TON Network. И хотя этот документ описывает точный формат блоков TON Blockchain и обсуждает условия

валидности блокчейна и сериализованные доказательства недействительности,¹ он не предоставляет деталей о сетевых протоколах, используемых для распространения этих блоков, кандидатов в блоки, сгруппированных блоков и доказательств недействительности.

Аналогично, этот документ не предоставляет полный исходный код смарт-контрактов мастерчайна, используемых для выбора валидаторов, изменения конфигурируемых параметров или получения их текущих значений, или наказания валидаторов за их неправильное поведение, хотя эти смарт-контракты составляют важную часть общего состояния блокчейна и нулевого блока мастерчайна. Вместо этого, документ описывает расположение этих смарт-контрактов и их формальные интерфейсы.² Исходный код этих смарт-контрактов будет предоставлен отдельно в виде загружаемых файлов с комментариями.

Обратите внимание, что текущая версия этого документа описывает предварительную тестовую версию TON Blockchain; некоторые мелкие детали, вероятно, изменятся до запуска в ходе разработки, тестирования и развертывания.

¹ На август 2018 этот документ не включает детальное описание сериализованных доказательств недействительности, так как они, вероятно, будут значительно изменены в ходе разработки программного обеспечения для валидаторов. Описываются только общие принципы проектирования условий консенсуса и сериализованных доказательств недействительности.

² Это не включено в настоящую версию этого документа, но будет представлено в отдельном приложении к будущей редакции.

Содержание

1. **Обзор**
 - 1.1. Все является пакетом ячеек
 - 1.2. Основные компоненты блока и состояние блокчейна
 - 1.3. Условия консенсуса
 - 1.4. Логическое время и интервалы логического времени
 - 1.5. Общее состояние блокчейна
 - 1.6. Конфигурируемые параметры и смарт-контракты
 - 1.7. Новые смарт-контракты и их адреса
 - 1.8. Модификация и удаление смарт-контрактов
 2. **Пересылка сообщений и гарантии доставки**
 - 2.1. Адреса сообщений и вычисление следующего перехода
 - 2.2. Hypercube Routing протокол
 - 2.3. Instant Hypercube Routing и комбинированные гарантии доставки
 3. **Сообщения, дескрипторы сообщений и очереди**
 - 3.1. Адрес, валюта и структура сообщения
 - 3.2. Дескрипторы входящих сообщений
 - 3.3. Очередь исходящих сообщений и дескрипторы
 4. **Аккаунты и транзакции**
 - 4.1. Аккаунты и их состояния
 - 4.2. Транзакции
 - 4.3. Описание транзакций
 - 4.4. Вызов смарт-контрактов в TVM
 5. **Структура блока**
 - 5.1. Структура блока шардчейна
 - 5.2. Структура блока мастерчейна
 - 5.3. Сериализация пакета ячеек
-
- A. **Криптография на эллиптических кривых**
 - A.1. Эллиптические кривые
 - A.2. Криптография Curve25519
 - A.3. Криптография Ed25519

1 Обзор

В этой главе представлен обзор основных функций и принципов проектирования TON Blockchain. Более подробная информация по каждой теме представлена в последующих главах.

1.1 Все является пакетом ячеек

Все данные в блоках и состоянии TON Blockchain представлены как коллекция ячеек (см. [3, 2.5]). Поэтому эта глава начинается с общего описания ячеек.

1.1.1. Ячейки TVM. Помните, что TON Blockchain, так же как и Виртуальная Машина TON (TVM; см. [4]), представляют все постоянно хранящиеся данные в виде *коллекции или пакета* так называемых ячеек. Каждая ячейка содержит до 1023 бит данных и до 4 ссылок на другие ячейки. Циклические ссылки на ячейки не допускаются, поэтому ячейки обычно организованы в *деревья ячеек* или, скорее, *ориентированные ациклические графы ячеек (DAGs)*.³ Любое значение абстрактного алгебраического (зависимого) типа данных должно быть представлено (сериализовано) как дерево ячеек. Точное представление значений абстрактного типа данных в виде дерева ячеек выражено с помощью схемы TL-B.⁴ Более полную информацию о различных видах ячеек можно найти в [4, 3.1].

1.1.2. Применение к блокам и состоянию TON Blockchain. Вышеизложенное особенно применимо к блокам и состоянию TON Blockchain, которые также являются значениями определенных (довольно сложных) зависимых алгебраических типов данных. Поэтому они сериализуются в соответствии с различными схемами TL-B (которые постепенно представлены в этом документе) и представлены как коллекция или пакет ячеек.

1.1.3. Структура отдельной ячейки. Каждая отдельная ячейка содержит до 1023 бит данных и до 4 ссылок на другие ячейки. Когда ячейка хранится в памяти, ее точное представление зависит от реализации. Однако существует стандартное представление ячеек, полезное, например, для сериализации ячеек для хранения файлов или передачи по сети. Это "стандартное представление" или "стандартная структура" CELLREPR(*c*) ячейки *c* состоит из следующего:

³ Полностью идентичные ячейки часто идентифицируются в памяти и на диске; это объясняет, почему деревья ячеек прозрачно преобразуются в DAGs ячеек. С этой точки зрения, DAG — это просто оптимизация хранения базового дерева ячеек, не имеющая значения для большинства рассмотрений.

⁴ См. [4, 3.3.3–4], где приводится и объясняется пример, рассматриваемый с более полным описанием.

- Два байта-дескриптора идут первыми, иногда обозначаются как d_1 и d_2 . Первый из этих байтов d_1 равен (в простейшем случае) количеству ссылок $0 \leq r \leq 4$ в ячейке. Второй байт-дескриптор d_2 кодирует длину l в битах части данных ячейки следующим образом: первые семь битов d_2 равны $\lfloor l / 8 \rfloor$, количество полных байтов данных, присутствующих в ячейке, в то время как последний бит d_2 является *тегом завершения*, равным единице, если l не делится на восемь. Таким образом,

$$d_2 = 2\lfloor l/8 \rfloor + [l \bmod 8 \neq 0] = \lfloor l/8 \rfloor + \lceil l/8 \rceil \quad (1)$$

где $[A]$ равно единице, когда условие A истинно, и нулю в противном случае.

- Далее следуют $\lfloor l / 8 \rfloor$ байтов данных. Это означает, что l битов данных ячейки разделены на группы по восемь, и каждая группа интерпретируется как целое восьмибитное big-endian число и сохраняется в байте. Если l не делится на восемь, добавляется один или несколько нулевых байтов (до шести) к битам данных, и тег завершения (наименее значимый бит байта дескриптора d_2) устанавливается.
- Наконец следуют r ссылок на другие ячейки. Каждая ссылка обычно представляется 32 байтами, содержащими SHA256-хэш ссылаемой ячейки, вычисленный, как объяснено ниже в [1.1.4](#).

Таким образом, стандартное представление CELLREPR(c) ячейки c с l битами данных и r ссылками имеет длину $2 + \lfloor l / 8 \rfloor + \lceil l / 8 \rceil + 32r$ байтов.

1.1.4. SHA256-хэш ячейки. SHA256-хэш ячейки c рекурсивно определяется как SHA256 стандартного представления CELLREPR(c) данной ячейки:

$$\text{HASH}(c) := \text{SHA256}(c) := \text{SHA256}(\text{CELLREPR}(c)) \quad (2)$$

Поскольку циклические ссылки на ячейки не допускаются (отношения между всеми ячейками должны составлять ориентированный ациклический граф или DAG), SHA256-хэш ячейки всегда четко определен.

Кроме того, поскольку SHA256 обычно считается устойчивым к коллизиям, мы предполагаем, что все ячейки, с которыми мы сталкиваемся, полностью определены их хэшами. В частности, ссылки на ячейку c полностью определяются хэшами ссылаемых ячеек, содержащихся в стандартном представлении CELLREPR(c).

1.1.5. Экзотические ячейки (exotic cells). Помимо *обычных (ordinary)* ячеек (также называемых *простыми (simple)* или *ячейками данных*), рассмотренных до сих пор, ячейки других типов, называемые *экзотическими ячейками*, иногда появляются в фактических представлениях блоков TON Blockchain и других структур данных. Их представление несколько отличается; они отличаются тем, что первый байт дескриптора $d_1 \geq 5$ (см. [\[4, 3.1\]](#)).

1.1.6. Внешние ссылочные ячейки. (*Внешние*) ссылочные ячейки, которые содержат 32-байтовый SHA256(c) «истинной» ячейки данных с вместо самой ячейки данных, являются одним из примеров экзотических ячеек. Эти ячейки могут использоваться в сериализации пакета ячеек, соответствующего блоку TON Blockchain, для ссылки на ячейки данных, отсутствующие в сериализации самого блока, но предполагаемые присутствующими где-то еще (например, в предыдущем состоянии блокчейна).

1.1.7. Прозрачность ссылочных ячеек в отношении большинства операций.

Большинство операций с ячейками не наблюдают никаких ссылок или других «экзотических» видов ячеек. Все такие ячейки при любой ссылке на ячейку прозрачно заменяются на ссылку на ячейку, на которую они ссылаются. Например, когда прозрачный хэш ячейки $\text{HASH}^b(c)$ рекурсивно вычисляется, хэш ссылочной ячейки устанавливается равным хэшу ссылаемой ячейки, а не хэшу стандартного представления ссылочной ячейки.

1.1.8. Прозрачный хэш и хэш представления ячейки. Таким образом, $\text{SHA256}^b(c) = \text{HASH}^b(c)$ является *прозрачным хэшем* ячейки c (или дерева ячеек с корнем в c).

Однако иногда нам нужно рассуждать о точном представлении дерева ячеек, присутствующего в блоке. Для этого определяется *хэш представления* $\text{HASH}^H(c)$, который не является прозрачным в отношении ссылочных ячеек и других экзотических типов ячеек. Мы часто говорим, что хэш представления c является «тем самым» хэшем c , потому что это наиболее часто используемый хэш ячейки.

1.1.9. Использование хэшей представления для подписей. Подписи являются отличным примером применения хэшей представления. Например:

- Валидаторы подписывают хэш представления блока, а не только его прозрачный хэш, потому что им нужно удостовериться, что блок действительно содержит требуемые данные, а не только некоторые внешние ссылки на них.
- Когда внешние сообщения подписываются и отправляются вне цепочки (например, клиентами, использующими приложение для инициирования транзакций в блокчейне), если в некоторых из этих сообщений могут присутствовать внешние ссылки, подписываются хэши представления этих сообщений.

1.1.10. Высшие хэши ячейки. В дополнение к прозрачным хэшам и хэшам представления ячейки c может быть определена последовательность *высших хэший* $\text{HASH}_i(c)$, $i = 1, 2, \dots$, которая в итоге стабилизируется на $\text{HASH}_\infty(c)$. (Более подробно можно ознакомиться в [4, 3.1].)

1.2 Основные компоненты блока и состояния блокчейна

В этом разделе кратко описываются основные компоненты блока и состояние блокчейна без углубления в детали.

1.2.1. The Infinite Sharding Paradigm (ISP), применяемая к блоку и состоянию шардчайна

Напомним, что согласно The Infinite Sharding Paradigm, каждый аккаунт может рассматриваться, как находящийся в своем отдельном аккаунтчейне (accountchain), и виртуальные блоки этих аккаунтчейнов затем группируются в блоки шардчайна для эффективности. В частности, состояние шардчайна состоит, грубо говоря, из состояний всех его аккаунтчейнов (т. е. всех аккаунтов, назначенных ему); аналогично блок шардчайна по существу состоит из коллекции виртуальных "блоков" для некоторых аккаунтов, назначенных этому шардчайну.⁵

Мы можем обобщить это следующим образом:

$$ShardState \approx Hashmap(n, AccountState) \quad (3)$$

$$ShardBlock \approx Hashmap(n, AccountBlock) \quad (4)$$

где n — это длина бита $account_id$, и $Hashmap(n, X)$ описывает частичную карту $2^n \rightarrow X$ от строк битов длины n к значениям типа X .

Напомним, что каждый шардчайн или, точнее, каждый блок шардчайна⁶ соответствует всем аккаунтчейнам, принадлежащим одному и тому же воркчейну (workchain) (т. е. имеют одинаковый $workchain_id = w$) и имеют $account_id$, начинающийся с того же бинарного префикса s , так что (w, s) полностью определяет шард. Следовательно, приведенные выше хэшмапы должны содержать только ключи, начинающиеся с префикса s .

Сейчас мы увидим, что приведенное выше описание является лишь приближением: состояние и блок шардчайна должны содержать дополнительные данные, которые не разделяются в соответствии с $account_id$, как предполагается в (3).

1.2.2. Разделенная и неразделенная часть блока и состояния шардчайна. Блок шардчайна и его состояние могут быть классифицированы на две отдельные части. Части с формой, продиктованной ISP, из (3) будут называться *разделенными (split)* частями блока и его состояния, в то время как оставшаяся часть будет называться *неразделенными (non-split)* частями.

⁵ Если с аккаунтом не связано никаких транзакций, соответствующий виртуальный блок пуст и опускается в блоке шардчайна.

⁶ Напомним, что TON Blockchain поддерживает *динамическое* шардингование, поэтому конфигурация шарда может изменяться от блока к блоку из-за объединения и разделения шардов. Следовательно, мы не можем просто сказать, что каждый шардчайн соответствует фиксированному набору аккаунтчейнов.

1.2.3. Взаимодействие с другими блоками и внешним миром. Глобальные и локальные условия консенсуса. Неразделенные части блока шардчайна и его состояния в основном связаны с взаимодействием этого блока с другими "соседними" блоками. Глобальные условия консенсуса блокчайна в целом сводятся к внутренним условиям консенсуса отдельных блоков, а также к внешним локальным условиям консенсуса между определенными блоками (см. [1.3](#)).

Большинство этих локальных условий консенсуса связаны с пересылкой сообщений между разными шардчайнами, транзакциями, затрагивающими более одного шардчайна, и гарантиями доставки сообщений. Однако другая группа локальных условий консенсуса связывает блок с его непосредственными предшественниками и последователями внутри шардчайна; например, начальное состояние блока обычно должно совпадать с конечным состоянием его непосредственного предшественника.⁷

1.2.4 Входящие и исходящие сообщения блока. Наиболее важными компонентами неразделенной части блока шардчайна являются следующие:

- *InMsgDescr* — Описание всех сообщений, «импортированных» в этот блок (т. е. либо обработанных транзакцией, включенной в блок, либо направленных в выходную очередь в случае транзитного сообщения, передающегося по пути, определенному Hypercube Routing).
- *OutMsgDescr* — Описание всех сообщений, «экспортированных» или «сгенерированных» блоком (т. е. либо сообщений, сгенерированных транзакцией, включенной в блок, либо транзитных сообщений, чье назначение не относится к текущему шардчайну, переданных из *InMsgDescr*).

1.2.5. Заголовок блока (Block header). Еще одной неразделенной частью блока шардчайна является заголовок блока, который содержит общую информацию, такую как (*w, s*) (т. е. *workchain_id* и общий бинарный префикс всех *account_ids*, назначенных текущему шардчайну), номер последовательности блока (*sequence number*, определяемый как наименьшее неотрицательное целое число, большее, чем номера последовательностей его предшественников), логическое время (*logical time*) и время генерации в формате unixtime (*generation unixtime*). Он также содержит хэш непосредственного предшественника блока (или двух его непосредственных предшественников в случае предшествующего события слияния шардчайнов), хэши начального и конечного состояний (т. е. состояний шардчайна непосредственно до и сразу после обработки текущего блока), а также хэш самого последнего известного блока мастерчайна на момент генерации блока шардчайна.

⁷ Это условие применяется, если есть точно один непосредственный предшественник (т. е. если событие объединения шардов не произошло непосредственно перед рассматриваемым блоком); в противном случае это условие становится более запутанным.

1.2.6. Подписи валидаторов, подписанные и неподписанные блоки. Описанный до сих пор блок является *неподписаным блоком*; он генерируется целиком и рассматривается валидаторами как единое целое. Когда валидаторы в итоге подписывают его, создается *подписанный блок*, состоящий из неподписанного блока вместе со списком подписей валидаторов (определенного хэша представления неподписанного блока, см. [1.1.9](#)). Этот список подписей также является неразделенной частью (подписанного) блока; однако, поскольку он находится за пределами неподписанного блока, он несколько отличается от других данных, хранящихся в блоке.

1.2.7. Очередь исходящих сообщений шардчайна. Аналогично, наиболее важной неразделенной частью состояния шардчайна является *OutMsgQueue*, очередь исходящих сообщений. Она содержит *недоставленные* сообщения, включенные в *OutMsgDescr*, либо последним блоком шардчайна, ведущим к этому состоянию, либо одним из его предшественников.

Изначально каждое исходящее сообщение включается в *OutMsgQueue*; оно удаляется из очереди только после того, как оно было включено в *InMsgDescr* блока "соседнего" шардчайна (следующего по Hypercube Routing), или было доставлено (т. е. появилось в *InMsgDescr*) своего конечного шардчайна назначения через Instant Hypercube Routing. В обоих случаях причина удаления сообщения из *OutMsgQueue* четко указана в *OutMsgDescr* блока, в котором произошло такое преобразование состояния.

1.2.8. Структура *InMsgDescr*, *OutMsgDescr* и *OutMsgQueue*. Все наиболее важные неразделенные структуры данных шардчайна, связанные с сообщениями, организованы как *хэшмапы* или *словари* (реализованные с помощью деревьев Патриция, сериализованных в дерево ячеек, как описано в [\[4, 3.3\]](#)), со следующими ключами:

- Описание входящего сообщения *InMsgDescr* использует 256-битный хэш сообщения в качестве ключа.
- Описание исходящего сообщения *OutMsgDescr* использует 256-битный хэш сообщения в качестве ключа.
- Очередь исходящих сообщений *OutMsgQueue* использует 352-битную конкатенацию 32-битного идентификатора целевого *workchain_id*, первых 64 битов целевого адреса *account_id* и 256-битного хэша сообщения в качестве ключа.

1.2.9. Разделенная часть блока: цепочки транзакций. Разделенная часть блока шардчайна состоит из хэшмапа, который сопоставляет некоторые аккаунты, назначенные шардчайну, "виртуальным блокам аккаунтчайна" *AccountBlock*, см. [\(3\)](#). Такой виртуальный блок аккаунтчайна состоит из последовательного списка *транзакций*, связанных с этим аккаунтом.

1.2.10. Описание транзакции. Каждая транзакция описывается в блоке экземпляром типа *Transaction*, который содержит в частности следующую информацию:

- Ссылка на ровно одно *входящее сообщение* (которое также должно присутствовать в *InMsgDescr*), *обработанное транзакцией*.
- Ссылки на несколько (возможно, ноль) *исходящих сообщений* (также присутствующих в *OutMsgDescr* и, скорее всего, включенных в *OutMsgQueue*), которые были *сгенерированы* транзакцией.

Транзакция состоит из вызова TVM (см. [4]) с кодом смарт-контракта, соответствующего данному аккаунту, загруженного в виртуальную машину, и с корневой ячейкой данных смарт-контракта, загруженной в регистр *c4* виртуальной машины. Само входящее сообщение передается в стек в качестве аргумента функции *main()* смарт-контракта, наряду с другими важными данными, такими как количество TON Grams и других определенных валют, привязанных к сообщению, адрес аккаунта отправителя, текущий баланс смарт-контракта и т. д.

В дополнение к информации выше экземпляр *Transaction* также содержит исходное и конечное состояния аккаунта (т. е. смарт-контракта), а также некоторые статистические данные выполнения TVM (использованный газ, цена газа, выполненные инструкции, созданные/удаленные ячейки, код завершения работы виртуальной машины и т. д.).

1.2.11. Разделенная часть состояния шардчайна: состояния аккаунтов. Напомним, что согласно (3), разделенная часть состояния шардчайна состоит из хэшмапа, сопоставляющего каждый "определенный" идентификатор аккаунта (принадлежащий данному шардчайну) с *состоянием* соответствующего аккаунта, заданным экземпляром типа *AccountState*.

1.2.12. Состояние аккаунта. Состояние аккаунта само по себе примерно состоит из следующих данных:

- Его *баланс* в Grams (и, возможно, в некоторых других определенных криптовалютах/токенах).
- *Код смарт-контракта* или хэш кода смарт-контракта, если он будет предоставлен (загружен) позже отдельным сообщением.
- Постоянные *данные смарт-контракта*, которые могут быть пустыми для простых смарт-контрактов. Это дерево ячеек, корень которого загружается в регистр *c4* во время выполнения смарт-контракта.
- *Статистика использования хранилища*, включая количество ячеек и байтов, хранящихся в постоянном хранилище смарт-контракта (т. е. внутри состояния блокчайна), и последний раз, когда плата за использование хранилища взималась с этого аккаунта.
- Необязательное *формальное описание интерфейса* (предназначенное для смарт-контрактов) и/или *публичная информация пользователя* (предназначенная в основном для людей и организаций).

Обратите внимание, что в TON Blockchain нет различия между "смарт-контрактом" и "аккаунтом". Вместо этого "простые" или "аккаунты-кошельки", как правило, используемые людьми и их приложениями-кошельками для простых переводов криптовалют, являются простыми смарт-контрактами со стандартным (общим) кодом и с

постоянными данными, состоящими из публичного ключа кошелька (или нескольких публичных ключей в случае мультиподписи; см. [1.7.6](#) для более подробной информации).

1.2.13. Блоки мастерчайна. В дополнение к блокам шардчайна и их состояниям TON Blockchain содержит блоки мастерчайна и состояние мастерчайна (также называемое *глобальным состоянием*). Блоки и состояние мастерчайна во многом похожи на блоки шардчайна и состояния, рассмотренные до сих пор, с некоторыми заметными отличиями:

- Мастерчайн не может быть разделен или объединен, поэтому блок мастерчайна обычно имеет только одного непосредственного предшественника. Единственным исключением является "нулевой блок мастерчайна", который отличается тем, что имеет номер последовательности, равный нулю; он не имеет предшественников и содержит начальную конфигурацию всего TON Blockchain (например, исходный набор валидаторов).
- Блоки мастерчайна содержат еще одну важную неразделенную структуру: *ShardHashes*, бинарное дерево со списком всех определенных шардчайнов вместе с хэшами последних блоков внутри каждого из перечисленных шардчайнов. Включение блока шардчайна в эту структуру делает блок шардчайна "каноническим" и позволяет другим блокам шардчайнов ссылаться на данные (например, исходящие сообщения), содержащиеся в блоке шардчайна.
- Состояние мастерчайна содержит глобальные параметры конфигурации всего TON Blockchain, такие как минимальные и максимальные цены на газ, поддерживаемые версии TVM, минимальная ставка для кандидатов в валидаторы, список альтернативных криптовалют, поддерживаемых в дополнение к Grams, общее количество выпущенных Grams и текущий набор валидаторов, ответственных за создание и подпись новых блоков, вместе с их публичными ключами.
- Состояние мастерчайна также содержит код смарт-контрактов, используемых для выбора последующих наборов валидаторов и для изменения глобальных параметров конфигурации. Код этих смарт-контрактов сам по себе является частью глобальных параметров конфигурации и может быть соответственно изменен. В этом отношении этот код (вместе с текущими значениями этих параметров) функционирует как "конституция" TON Blockchain. Он первоначально устанавливается в нулевом блоке мастерчайна.
- Через мастерчайн не проходят транзитные сообщения: каждое входящее сообщение должно иметь место назначения внутри мастерчайна, а каждое исходящее сообщение должно иметь источник внутри мастерчайна.

1.3 Условия консенсуса

В дополнение к структурам данных, содержащимся в блоке и в состоянии блокчайна, которые сериализуются в пакеты ячеек в соответствии с определенными схемами TL-B, объясняемыми более подробно позже (см. главы [3–5](#)), важным компонентом структуры блокчайна являются *условия консенсуса* между данными, хранимыми внутри одного или

разных блоков (как упоминалось в [1.2.3](#)). В этом разделе подробно описывается функция условий консенсуса в блокчейне.

1.3.1. Выражение условий консенсуса. В принципе, зависимые типы данных (например, те, которые используются в TL-B) могут быть использованы не только для описания сериализации данных блока, но и для выражения условий, наложенных на компоненты таких типов данных. (Например, можно определить тип данных *OrderedIntPair*, с парами целых чисел (x, y), таких что $x < y$, в качестве значений.) Однако TL-B в настоящее время недостаточно выразителен, чтобы закодировать все условия консенсуса, которые нам нужны, поэтому мы выбираем полуформализованный подход в этом тексте. В будущем мы можем представить последующую полную формализацию в подходящем доказательном помощнике, таком как Соq.

1.3.2. Важность условий консенсуса. Условия консенсуса в итоге важны не менее, чем "неограниченные" структуры данных, на которые они наложены, особенно в контексте блокчейна. Например, условия консенсуса обеспечивают, что состояние аккаунта не изменяется между блоками и может измениться только внутри блока в результате транзакции. Таким образом, условия консенсуса обеспечивают безопасное хранение балансов криптовалют и другой информации внутри блокчейна.

1.3.3. Виды условий консенсуса. Существует несколько видов условий консенсуса, наложенных на TON Blockchain:

- *Глобальные условия* — Выражают инварианты по всему TON Blockchain. Например, *гарантии доставки сообщений*, которые утверждают, что каждое сгенерированное сообщение должно быть доставлено до своего целевого аккаунта и доставлено ровно один раз, являются частью глобальных условий.
- *Внутренние (локальные) условия* — Выражают условия, наложенные на данные, хранимые внутри одного блока. Например, каждая транзакция, включенная в блок (т. е. присутствующая в списке транзакций какого-либо аккаунта), обрабатывает ровно одно входящее сообщение; это входящее сообщение также должно быть указано в структуре *InMsgDescr* блока.
- *Внешние (локальные) условия* — Выражают условия, наложенные на данные различных блоков, обычно принадлежащих одному или соседним шардчейнам (по отношению к Hypercube Routing). Таким образом, внешние условия бывают нескольких видов:
 - *Условия предшественника/последователя* — Выражают условия, наложенные на данные некоторого блока и его непосредственного предшественника или (в случае предшествующего события слияния шардчейнов) двух непосредственных предшественников. Наиболее важным из этих условий является то, что начальное состояние для блока шардчейна должно совпадать с конечным состоянием шардчейна его непосредственного предшественника, при условии, что между ними не происходило события разделения/слияния шардчейнов.

- *Условия мастерчайна/шардчайна* — Выражают условия, наложенные на блок шардчайна и блок мастерчайна, которые ссылаются на него в своем списке *ShardHashes* или упоминаются в заголовке блока шардчайна.
- *Условия соседнего (блока)* — Выражают отношения между блоками соседних шардчайнов по отношению к Hypercube Routing. Наиболее важные из этих условий выражают связь между *InMsgDescr* блока и *OutMsgQueue* состояния соседнего блока.

1.3.4. Декомпозиция глобальных и локальных условий в более простые локальные условия. Глобальные условия консенсуса, такие как гарантии доставки сообщений, действительно необходимы для правильной работы блокчайна; однако их трудно обеспечить и проверить напрямую. Поэтому мы вводим множество более простых локальных условий консенсуса, которые легче обеспечить и проверить, так как они касаются только одного блока или, возможно, двух смежных блоков. Эти локальные условия выбраны таким образом, чтобы желаемые глобальные условия были логическими следствиями (конъюнкцией) всех локальных условий. В этом отношении мы говорим, что глобальные условия были "декомпозированы" на более простые локальные условия.

Иногда локальное условие все же оказывается слишком сложным для обеспечения или проверки. В этом случае оно декомпозируется дальше, на еще более простые локальные условия.

1.3.5. Декомпозиция может требовать дополнительных структур данных и дополнительных внутренних условий консенсуса. Декомпозиция условия в более простые локальные условия консенсуса иногда требует введения дополнительных структур данных. Например, структура *InMsgDescr* явно перечисляет все входящие сообщения, обработанные в блоке, даже если этот список мог быть получен путем сканирования списка всех транзакций, присутствующих в блоке. Однако *InMsgDescr* значительно упрощает условия для соседних блоков, связанные с пересылкой и маршрутизацией сообщений, что итоге приводит к глобальным гарантиям доставки сообщений.

Обратите внимание, что введение таких дополнительных структур данных является своего рода "денормализацией базы данных" (т. е. приводит к некоторой избыточности или к тому, что некоторые данные присутствуют более одного раза), и поэтому необходимо наложить больше внутренних условий консенсуса (например, если некоторые данные теперь присутствуют в двух копиях, мы должны требовать, чтобы эти две копии совпадали). Например, как только мы вводим *InMsgDescr* для облегчения пересылки сообщений между шардчайнами, нам нужно ввести внутренние условия консенсуса, связывающие *InMsgDescr* со списком транзакций того же блока.

1.3.6. Корректные условия сериализации. Помимо высокоуровневых внутренних условий консенсуса, которые рассматривают содержимое блока как значение абстрактного типа данных, существуют некоторые более низкоуровневые внутренние условия консенсуса, называемые "(корректными) условиями сериализации", которые обеспечивают, что дерево ячеек, присутствующее в блоке, действительно является

правильной сериализацией значения ожидаемого абстрактного типа данных. Такие условия сериализации могут быть автоматически сгенерированы из схемы TL-B, описывающей абстрактный тип данных и его сериализацию в дерево ячеек.

Обратите внимание, что условия сериализации представляют собой набор взаимно рекурсивных предикатов на ячейках или срезах ячеек. Например, если значение типа A состоит из 32-битного магического числа m_A , 64-битного целого числа l и двух ссылок на ячейки, содержащих значения типов B и C соответственно, то корректное условие сериализации для значений типа A потребует, чтобы ячейка или срез ячейки содержали ровно 96 бит данных и две ссылки на ячейки r_1 и r_2 , с дополнительным требованием, чтобы первые 32 бита данных содержали m_A и чтобы две ячейки, на которые ссылаются r_1 и r_2 , удовлетворяли условиям сериализации для значений типов B и C соответственно.

1.3.7. Конструктивное устранение кванторов существования. Локальные условия, которые могут быть наложены, иногда являются *неконструктивными*, что означает, что они не обязательно содержат объяснение того, почему они истинны. Типичным примером такого условия C является следующее условие

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

«для любого x из X существует y из Y такой, что выполняется условие $A(x, y)$ ». Даже если мы знаем, что C истинно, у нас нет способа быстро найти y из Y , такой что $A(x, y)$, для данного $x : X$. В результате проверка C может занять много времени.

Чтобы упростить проверку локальных условий, их делают *конструктивными* (т. е. проверяемыми за ограниченное время) путем добавления некоторых *свидетельских* структур данных. Например, условие C из (5) может быть преобразовано добавлением новой структуры данных $f : X \rightarrow Y$ (отображение из X в Y) и наложением следующего условия C' вместо него:

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

Конечно, "свидетельское" значение $f(x) : Y$ может быть включено внутрь (измененной) структуры данных X вместо того, чтобы храниться в отдельной таблице f .

1.3.8. Пример: условие консенсуса для *InMsgDescr*. Например, условие консенсуса между $X := InMsgDescr$, списком всех входящих сообщений, обработанных в блоке, и $Y := Transactions$, списком всех транзакций, присутствующих в блоке, является вышеуказанным типом:

"Для любого входящего сообщения x , присутствующего в $InMsgDescr$, транзакция y должна присутствовать в блоке, так что y обрабатывает x ".⁸ Процедура устранения \exists , описанная в **1.3.7**, приводит нас к необходимости ввести дополнительное поле в дескрипторы входящих сообщений $InMsgDescr$, содержащее ссылку на транзакцию, в которой сообщение действительно обработано.

1.3.9. Конструктивное устранение логических дизъюнкций. Аналогично преобразованию, описанному в **1.3.7**, условие

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

"для любого x из X хотя бы одно из $A_1(x)$ или $A_2(x)$ истинно", может быть преобразовано в функцию $i : X \rightarrow \mathbf{2} = \{1, 2\}$ и новое условие

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

Это особый случай устранения кванторов существования, рассмотренных ранее для $Y = \mathbf{2} = \{1, 2\}$. Это может быть полезно, когда $A_1(x)$ и $A_2(x)$ являются сложными условиями, которые нельзя быстро проверить, поэтому полезно заранее знать, какое из них на самом деле истинно.

Например, $InMsgDescr$, рассмотренный в **1.3.8**, может содержать как сообщения, обработанные в блоке, так и транзитные сообщения. Мы можем ввести поле в описании входящего сообщения, чтобы указать, является ли сообщение транзитным, и, в последнем случае, включить свидетельское поле для транзакции, обрабатывающей сообщение.

1.3.10. Конструтивизация условий. Этот процесс устранения неконструтивных логических связок \exists (квантор существования) и (иногда) \vee (логическая дизъюнкция) путем введения дополнительных структур данных и полей, то есть процесс превращения условия в конструктивное будет называться *конструтивизацией*. Если довести этот процесс до теоретического предела, он приводит к логическим формулам, содержащим только всеобщие кванторы и логические конъюнкции за счет добавления некоторых свидетельских полей в определенные структуры данных.

1.3.11. Условия валидности для блока. В итоге все внутренние условия для блока, вместе с локальными условиями предшественника и соседа, касающимися этого блока и другого ранее генерированного блока, составляют *условия валидности* для блока шардчайна или мастерчайна. Блок является *валидным*, если он удовлетворяет условиям валидности. Ответственность за генерацию валидных блоков, а также за проверку валидности блоков, созданных другими валидаторами, лежит на валидаторах.

⁸ Этот пример несколько упрощен, так как не учитывает наличие транзитных сообщений в $InMsgDescr$, которые не обрабатываются никакой явной транзакцией.

1.3.12. Свидетельства недействительности блока. Если блок не удовлетворяет всем условиям валидности C_1, \dots, C_n (т. е. конъюнкции $V := \wedge_i C_i$ условий валидности), он является недействительным. Это означает, что он удовлетворяет "условию недействительности" $\neg V = \vee_i \neg C_i$. Если все C_i и, следовательно, V были "конструктивизированы" в смысле, описанном в [1.3.10](#), так что они содержат только логические конъюнкции и всеобщие кванторы (и простые атомарные утверждения), то $\neg V$ содержит только логические дизъюнкции и кванторы существования. Тогда может быть определена конструктивизация $\neg V$, которая будет включать *свидетельство недействительности*, начиная с индекса i конкретного условия валидности C_i , которое не выполнено.

Такие свидетельства недействительности могут также быть сериализованы и представлены другим валидаторам или внесены в мастерчейн, чтобы доказать, что конкретный блок или кандидат в блок на самом деле недействителен. Поэтому построение и сериализация свидетельств недействительности являются важной частью проектирования блокчайна Proof-of-Stake (PoS).⁹

1.3.13. Минимизация размера свидетельств. Важным аспектом для проектирования локальных условий, их декомпозиции в более простые условия и их конструктивизации является упрощение проверки каждого условия настолько, насколько это возможно. Однако еще одним требованием является минимизация размера свидетельств как для условия (чтобы размер блока не увеличивался слишком сильно в процессе конструктивизации), так и для его отрицания (чтобы доказательства недействительности имели ограниченный размер, что упрощает их проверку, передачу и включение в мастерчейн). Эти два принципа проектирования иногда противоречат друг другу, и тогда нужно искать компромисс.

1.3.14. Минимизация размера доказательств Меркла. Изначально условия консенсуса предназначены для обработки стороной, у которой уже есть все соответствующие данные (например, все блоки, упомянутые в условии). Однако в некоторых случаях они должны быть проверены стороной, у которой нет всех блоков, но есть только их хэши. Например, предположим, что доказательство недействительности блока дополнено подписью валидатора, который подписал недействительный блок (и, следовательно, должен быть наказан). В этом случае подпись будет содержать только хэш неправильно подписанного блока; сам блок придется восстанавливать из другого места до проверки доказательства недействительности блока.

Компромисс между предоставлением только хэша предположительно недействительного блока и предоставлением всего недействительного блока вместе со свидетельством недействительности заключается в дополнении свидетельства недействительности доказательством Меркла, начиная с хэша блока (т. е. корневой ячейки блока). Такое доказательство включало бы все ячейки, на которые ссылается свидетельство недействительности, вместе со всеми ячейками на путях от этих ячеек до корневых ячеек и хэшами их родственных ячеек.

⁹ Интересно отметить, что эту часть работы можно сделать почти автоматически.

Тогда доказательство недействительности становится достаточным, чтобы обеспечить обоснование для наказания валидатора. Например, предложенное выше доказательство недействительности может быть представлено смарт-контракту, находящемуся в мастерчайне, который наказывает валидаторов за неправильное поведение.

Поскольку такое доказательство недействительности должно быть дополнено доказательством Меркля, имеет смысл писать условия консенсуса таким образом, чтобы доказательства Меркля для их отрицаний были как можно меньше. В частности, каждое индивидуальное условие должно быть максимально "локальным" (т. е. включать минимальное количество ячеек). Это также оптимизирует время проверки доказательства недействительности.

1.3.15. Сводные данные для внешних условий. Когда валидатор предлагает неподписанный блок другим валидаторам шардчайна, эти другие валидаторы должны проверить действительность этого кандидата в блок, т. е. проверить, что он удовлетворяет всем внутренним и внешним локальным условиям консенсуса. В то время как внутренние условия не требуют никаких дополнительных данных, кроме самого кандидата в блок, внешние условия нуждаются в некоторых других блоках или по крайней мере в некоторой информации из этих блоков. Такая дополнительная информация может быть извлечена из этих блоков вместе со всеми ячейками на путях от ячеек, содержащих требуемую дополнительную информацию, до корневой ячейки соответствующих блоков и хэшами родственных ячеек, находящихся на каждом из этих путей, чтобы представить доказательство Меркля, которое может быть обработано без знания самих ссылочных блоков.

Эта дополнительная информация, называемая сводными данными, сериализуется в виде набора ячеек и представляется валидатором вместе с неподписаным кандидатом в блок. Кандидат в блок вместе со сводными данными называется сводным блоком.

1.3.16. Условия для сводного блока. Внешние условия консенсуса для кандидата в блок таким образом (автоматически) преобразуются во *внутренние* условия консенсуса для сводного блока, что значительно упрощает и ускоряет их проверку другими валидаторами. Однако некоторые данные, такие как конечное состояние непосредственного предшественника проверяемого блока, не включаются в сводные данные. Вместо этого предполагается, что все валидаторы хранят локальную копию этих данных.

1.3.17. Условия представления и хэши представления. Обратите внимание, что, как только доказательства Меркля включены в сводный блок, условия консенсуса должны учитывать, какие данные (т. е. какие ячейки) фактически присутствуют в сводном блоке, а не только ссылаться на них по их хэшам. Это приводит к новой группе условий, называемых *условиями представления*, которые должны уметь различать внешнюю ссылку на ячейку (обычно представленную ее 256-битным хэшем) и саму ячейку. Валидатор может быть наказан за предложение сводного блока, который не содержит всех ожидаемых сводных данных внутри, даже если сам кандидат в блок является допустимым.

Это также приводит к использованию хэшей представления вместо прозрачных хэшей для сводных блоков.

1.3.18. Проверка при отсутствии сводных данных. Обратите внимание, что блок все равно должен быть проверяемым в отсутствие сводных данных; в противном случае никто, кроме валидаторов, не сможет проверить ранее зафиксированный блок своими силами. В частности, свидетельства не могут быть включены в сводные данные: они должны находиться в самом блоке. Сводные данные должны содержать только некоторые части соседних блоков, на которые ссылаются в основном блоке, вместе с подходящими доказательствами Меркла, которые могут быть восстановлены любым, у кого есть сами ссылочные блоки.

1.3.19. Включение доказательств Меркла в сам блок. Обратите внимание, что в некоторых случаях доказательства Меркла должны быть встроены в сам блок, а не только в сводные данные. Например:

- Во время Instant Hypercube Routing (IHR) сообщение может быть включено непосредственно в *InMsgDescr* блока целевого шардчайна без прохождения по всему пути вдоль ребер гиперкуба. В этом случае доказательство Меркла существования сообщения в *OutMsgDescr* блока исходного шардчайна должно быть включено в *InMsgDescr* вместе с самим сообщением.
- Доказательство недействительности или другое доказательство неправильного поведения валидатора может быть зафиксировано в мастерчейне путем включения его в тело сообщения, отправленного специальному смарт-контракту. В этом случае доказательство недействительности должно включать некоторые ячейки вместе с доказательством Меркла, которое должно быть включено в тело сообщения.
- Аналогично, смарт-контракт, определяющий платежный канал или другой вид сайдчайна (*said-chain*), может принимать сообщения о финализации или доказательства неправильного поведения, содержащие подходящие доказательства Меркла.
- Конечное состояние шардчайна не включается в блок шардчайна. Вместо этого включаются только те ячейки, которые были изменены; те ячейки, которые унаследованы из старого состояния, ссылаются на их хэши вместе с подходящими доказательствами Меркла, состоящими из ячеек на пути от корня старого состояния к ячейкам старого состояния, на которые ссылаются.

1.3.20. Условия для обработки неполных данных. Как мы видели, необходимо включать неполные данные и доказательства Меркла в тело некоторых сообщений, содержащихся в блоке, и в состояние. Эта необходимость отражается в некоторых дополнительных условиях представления, а также в положениях для сообщений (и по расширению, деревьев ячеек, обрабатываемых TVM) содержать неполные данные (внешние ссылки на ячейки и доказательства Меркла). В большинстве случаев такие внешние ссылки на ячейки содержат только 256-битный SHA256 хэш ячейки вместе с флагом; если смарт-контракт пытается инспектировать содержимое такой ячейки с помощью примитива *CToS* (например, для десериализации), то вызывается исключение. Однако внешняя ссылка на такую ячейку может быть сохранена в постоянном хранилище

смарт-контракта, и могут быть вычислены как прозрачные хэши, так и хэши представления такой ячейки.

1.4 Логическое время и интервалы логического времени

Этот раздел более подробно рассматривает так называемое *логическое время*, которое широко используется в TON Blockchain для пересылки сообщений и гарантий доставки сообщений, среди прочих целей.

1.4.1. Логическое время. Компонент TON Blockchain, который также играет важную роль в доставке сообщений, это *логическое время*, обычно обозначаемое как LT. Это неотрицательное 64-битное целое число, назначаемое определенным событиям примерно следующим образом:

Если событие e логически зависит от событий e_1, \dots, e_n , то $\text{LT}(e)$ — это наименьшее неотрицательное число, большее, чем все $\text{LT}(e_i)$.

В частности, если $n = 0$ (т. е. если e не зависит ни от каких предыдущих событий), то $\text{LT}(e) = 0$.

1.4.2. Ослабленный вариант логического времени. В некоторых случаях мы ослабляем определение логического времени, требуя только, чтобы

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

без необходимости, чтобы $\text{LT}(e)$ было наименьшим неотрицательным числом с этим свойством. В таких случаях мы можем говорить об ослабленном логическом времени, в отличие от строгого логического времени, определенного выше (см. 1.4.1). Обратите внимание, однако, что условие (9) является фундаментальным свойством логического времени и не может быть дополнительно ослаблено.

1.4.3. Интервалы логического времени. Имеет смысл назначать некоторым событиям или наборам событий C интервал логического времени $\text{LT}^*(C) = [\text{LT}^-(C), \text{LT}^+(C)]$, что означает, что набор событий C произошел в указанном "интервале" логического времени, где $\text{LT}^-(C) < \text{LT}^+(C)$ являются некоторыми целыми числами (на практике 64-битными целыми числами). В этом случае можно сказать, что C начинается в логическое время $\text{LT}^-(C)$ и заканчивается в логическое время $\text{LT}^+(C)$.

По умолчанию, мы предполагаем, что $\text{LT}^+(e) = \text{LT}(e) + 1$ и $\text{LT}^-(e) = \text{LT}(e)$ для простых или "атомарных" событий, предполагая, что они занимают ровно одну единицу логического времени.

В общем случае, если у нас есть одно значение $\text{LT}(C)$ и интервал логического времени $\text{LT}^*(C) = [\text{LT}^-(C), \text{LT}^+(C)]$, мы всегда требуем, чтобы

$$\text{LT}(C) \in [\text{LT}^-(C), \text{LT}^+(C)) \quad (10)$$

или, эквивалентно,

$$\text{LT}^-(C) \leq \text{LT}(C) < \text{LT}^+(C) \quad (11)$$

В большинстве случаев мы выбираем $\text{LT}(C) = \text{LT}^-(C)$.

1.4.4. Требования к интервалам логического времени. Три основных требования к интервалам логического времени:

- $0 \leq \text{LT}^-(C) < \text{LT}^+(C)$ — это неотрицательные целые числа для любой коллекции событий C .
- Если $e' \prec e$ (т. е. если атомарное событие e логически зависит от другого атомарного события e'), тогда $\text{LT}^*(e') < \text{LT}^*(e)$ (т. е. $\text{LT}^+(e') \leq \text{LT}^-(e)$).
- Если $C \supset D$ (т. е. если коллекция событий C содержит другую коллекцию событий D), тогда $\text{LT}^*(C) \supset \text{LT}^*(D)$, т. е.

$$\text{LT}^-(C) \leq \text{LT}^-(D) < \text{LT}^+(D) \leq \text{LT}^+(C) \quad (12)$$

В частности, если C состоит из атомарных событий e_1, \dots, e_n , тогда

$$\text{LT}^-(C) \leq \inf_i \text{LT}^-(e_i) \leq \inf_i \text{LT}(e_i) \text{ и } \text{LT}^+(C) \geq \sup_i \text{LT}^+(e_i) \geq 1 + \sup_i \text{LT}(e_i).$$

1.4.5. Строгие или минимальные интервалы логического времени. Можно назначить любой конечной коллекции атомарных событий $E = \{e\}$, связанных отношением причинности (частичным порядком) \prec , и всем подмножествам $C \subseteq E$ **минимальные** интервалы логического времени. То есть, среди всех назначений интервалов логического времени, удовлетворяющих условиям, перечисленным в **1.4.4**, мы выбираем то, у которого $\text{LT}^+(C) - \text{LT}^-(C)$ минимально возможное, и если существует несколько назначений с этим свойством, мы выбираем то, у которого минимально и $\text{LT}^-(C)$.

Такое назначение можно достичь, сначала назначив логическое время $\text{LT}(e)$ всем атомарным событиям $e \in E$, как описано в **1.4.1**, затем установив $\text{LT}^-(C) := \inf_{e \in C} \text{LT}(e)$ и $\text{LT}^+(C) := 1 + \sup_{e \in C} \text{LT}(e)$ для любого $C \subseteq E$.

В большинстве случаев, когда нам нужно назначить интервалы логического времени, мы используем минимальные интервалы логического времени, описанные выше.

1.4.6. Логическое время в TON Blockchain. TON Blockchain присваивает логическое время и логические временные интервалы некоторым своим компонентам.

Например, каждому исходящему сообщению, созданному в транзакции, присваивается *логическое время создания*; для этой цели создание исходящего сообщения рассматривается как атомарное событие, логически зависящее от предыдущего сообщения, созданного той же транзакцией, а также от предыдущей транзакции того же аккаунта, от входящего сообщения, обработанного той же транзакцией, и от всех событий, содержащихся в блоках, на которые ссылаются хэши, содержащиеся в том же блоке с этой транзакцией. В результате, *исходящие сообщения, созданные одним и тем же смарт-контрактом, имеют строго возрастающее логическое время создания*. Сама транзакция рассматривается как набор атомарных событий и ей присваивается логический временной интервал (см. [4.2.1](#) для более точного описания).

Каждый блок представляет собой набор событий создания транзакций и сообщений, поэтому ему присваивается логический временной интервал, явно указанный в заголовке блока.

1.5. Общее состояние блокчейна

Этот раздел об общем состоянии TON Blockchain, а также о состояниях отдельных шардчейнов и мастерчайна. Например, точное определение состояния соседних шардчейнов становится важным для правильной формализации условия консенсуса, утверждающего, что валидаторы для шардчайна должны импортировать самые старые сообщения из объединения *OutMsgQueue*, взятые из состояний всех соседних шардчейнов (см. [2.2.5](#)).

1.5.1. Общее состояние, определенное блоком мастерчайна. Каждый блок мастерчайна содержит список всех текущих активных шардов и последних блоков для каждого из них. В этом отношении *каждый блок мастерчайна определяет соответствующее общее состояние TON Blockchain, так как он фиксирует состояние каждого шардчайна и мастерчайна*.

Важное требование, накладываемое на этот список последних блоков для всех шардчейнов, заключается в том, что, если блок мастерчайна B перечисляет S как последний блок некоторого шардчайна, и более новый блок мастерчайна B' с B в числе его предшественников перечисляет S' как последний блок того же шардчайна, то S должен быть одним из предшественников S' .¹⁰ Это условие определяет общее состояние TON Blockchain, определенное последующим блоком мастерчайна B' , совместимым с общим состоянием, определенным предыдущим блоком B .

¹⁰ Чтобы правильно выразить это условие при динамическом шардинге, необходимо зафиксировать некоторую учетную запись ζ и рассмотреть последние блоки S и S' шардчейнов, содержащих ζ в конфигурациях шардов как B , так и B' , поскольку шарды, содержащие ζ могут отличаться B и B' .

1.5.2. Общее состояние, определяемое блоком шардчейна. Каждый блок шардчейна содержит хэш самого последнего блока мастерчейна в своем заголовке. Следовательно, все блоки, на которые указывает этот блок мастерчейна, вместе с их предшественниками, считаются «известными» или «видимыми» для блока шардчейна, и никакие другие блоки для него не видны, за исключением его предшественников внутри его собственного шардчейна.

В частности, когда мы говорим, что блок должен импортировать в свой InMsgDescr сообщения из OutMsgQueue состояний всех соседних шардчейнов, это означает, что именно блоки других шардчейнов, видимые для этого блока, должны быть учтены, и в то же время блок не может содержать сообщения из «невидимых» блоков, даже если они в остальном правильные.

1.6. Конфигурируемые параметры и смарт-контракты

Напомним, что в TON Blockchain есть несколько так называемых «конфигурируемых параметров» (см. [3]), которые являются либо определенными значениями, либо определенными смарт-контрактами, находящимися в мастерчейне. Этот раздел о хранении и доступе к этим конфигурируемым параметрам.

1.6.1. Примеры конфигурируемых параметров. К свойствам блокчайна, контролируемым конфигурируемыми параметрами, относятся:

- Минимальный взнос для валидаторов.
- Максимальный размер группы избранных валидаторов.
- Максимальное количество блоков, за которые отвечает одна и та же группа валидаторов.
- Процесс выборов валидаторов.
- Процесс наказания валидаторов.
- Текущий активный и следующий избранный набор валидаторов.
- Процесс изменения конфигурируемых параметров, а также адрес смарт-контракта γ , ответственного за хранение значений конфигурируемых параметров и за изменение их параметров.

1.6.2. Местоположение значений конфигурируемых параметров. Конфигурируемые параметры сохраняются в постоянных данных специального конфигурационного смарт-контракта γ , находящегося в мастерчейне TON Blockchain. Более точно, первая ссылка корневой ячейки постоянных данных этого смарт-контракта является словарем, сопоставляющим 64-битные ключи (номера параметров) со значениями соответствующих параметров; каждое значение сериализовано в срез ячейки в зависимости от типа этого значения. Если значение является "смарт-контрактом" (обязательно находящимся в мастерчейне), вместо этого используется его 256-битный адрес аккаунта.

1.6.3. Быстрый доступ через заголовок блоков мастерчайна. Чтобы упростить доступ к текущим значениям конфигурируемых параметров и сократить доказательства Меркла, содержащие ссылки на них, заголовок каждого блока мастерчайна содержит адрес смарт-

контракта γ . Он также содержит прямую ссылку-ячейку на словарь, содержащий все значения конфигурируемых параметров, который находится в постоянных данных γ . Дополнительные условия консенсуса гарантируют, что эта ссылка совпадает с той, которая получена при осмотре конечного состояния смарт-контракта γ .

1.6.4. Получение значений конфигурируемых параметров get-методами.

Конфигурационный смарт-контракт γ предоставляет доступ к некоторым конфигурируемым параметрам посредством get-методов. Эти специальные методы смарт-контракта не меняют его состояние, а возвращают требуемые данные в стек TVM.

1.6.5. Получение значений конфигурируемых параметров get-сообщениями.

Аналогично, конфигурационный смарт-контракт γ может определить некоторые «обычные» методы (т. е. специальные входящие сообщения) для запроса значений определенных конфигурируемых параметров, которые будут отправлены в исходящих сообщениях, создаваемых транзакцией, обрабатывающей такое входящее сообщение. Это может быть полезно для некоторых фундаментальных смарт-контрактов, которым необходимо знать значения определенных конфигурируемых параметров.

1.6.6. Значения, полученные get-методами, могут отличаться от тех, которые получены через заголовок блока. Обратите внимание, что состояние конфигурационного смарт-контракта γ , включая значения конфигурируемых параметров, может изменяться несколько раз в блоке мастерчайна, если в этом блоке обрабатывается несколько транзакций смарт-контрактом γ . В результате значения, полученные с помощью get-методов γ или отправки get-сообщений к γ , могут отличаться от тех, которые можно получить, проверив ссылку в заголовке блока (см. [1.6.3](#)), что относится к **конечному** состоянию конфигурируемых параметров в блоке.

1.6.7. Изменение значений конфигурируемых параметров. Процедура изменения значений конфигурируемых параметров определена в коде смарт-контракта γ . Для большинства конфигурируемых параметров, называемых обычными, любой валидатор может предложить новое значение, отправив специальное сообщение с номером параметра и его предложенным значением к γ . Если предложенное значение подтверждается, смарт-контрактом собираются дальнейшие сообщения о голосовании от валидаторов, и если более двух третей как текущего, так и следующего набора валидаторов поддерживает предложение, значение изменяется.

Некоторые параметры, такие как текущий набор валидаторов, не могут быть изменены этим способом. Вместо этого текущая конфигурация содержит параметр с адресом смарт-контракта v , ответственного за выборы следующего набора валидаторов, и смарт-контракт γ принимает сообщения только от этого смарт-контракта v для изменения конфигурируемого параметра, содержащего текущий набор валидаторов.

1.6.8. Изменение процедуры выборов валидаторов. Если процедура выборов валидаторов должна быть изменена, это можно сделать, сначала включив в мастерчайн новый смарт-контракт выборов валидаторов, а затем изменив обычный конфигурируемый параметр, содержащий адрес v смарт-контракта выборов валидаторов. Это потребует, чтобы две трети валидаторов приняли предложение в голосовании, как описано выше в [1.6.7](#).

1.6.9. Изменение процедуры изменения конфигурируемых параметров. Аналогично, адрес конфигурационного смарт-контракта сам по себе является конфигурируемым параметром и может быть изменен также. Таким образом, самые фундаментальные параметры и смарт-контракты TON Blockchain могут быть изменены в любом направлении, согласованном квалифицированным большинством валидаторов.

1.6.10. Начальные значения настраиваемых параметров. Начальные значения большинства конфигурируемых параметров появляются в нулевом блоке мастерчайна как часть начального состояния мастерчайна, что явно присутствует без пропусков в этом блоке. Код всех фундаментальных смарт-контрактов также присутствует в начальном состоянии. Таким образом, оригинальная "конституция" и конфигурация TON Blockchain, включая первоначальный набор валидаторов, становятся явными в нулевом блоке.

1.7 Новые смарт-контракты и их адреса

Этот раздел о создании и инициализации новых смарт-контрактов, в частности происхождение их начального кода, постоянные данные и баланс. Также о назначении адресов аккаунтов новым смарт-контрактам.

1.7.1. Описание валидно только для мастерчайна и базового воркчайна. Механизмы создания новых смарт-контрактов и присвоения им адресов, описанные в этом разделе, валидны только для базового воркчайна и мастерчайна. Другие воркчайны могут определять свои собственные механизмы для решения этих проблем.

1.7.2. Перевод криптовалюты на неинициализированные аккаунты. Прежде всего, можно отправлять сообщения, включая сообщения несущие значение, на ранее не упомянутые аккаунты. Если входящее сообщение поступает в воркчайн с адресом назначения η , соответствующим неопределенному аккаунту, оно обрабатывается транзакцией так, как если бы код смарт-контракта был пустым (т. е. состоящим из неявного RET). Если сообщение несет значение, это приводит к созданию "неинициализированного аккаунта", который может иметь ненулевой баланс (если были отправлены сообщения, несущие значение),¹¹ но не имеет кода и данных. Поскольку даже неинициализированный аккаунт занимает некоторое постоянное хранилище (необходимое для хранения его баланса), время от времени будут взиматься небольшие платежи за постоянное хранилище с баланса аккаунта, пока он не станет отрицательным.

1.7.3. Инициализация смарт-контрактов с помощью сообщений-конструкторов.

Аккаунт или смарт-контракт создается путем отправки специального *сообщения-конструктора* M на его адрес η . Тело такого сообщения содержит дерево ячеек с начальным кодом смарт-контракта (которое может быть заменено его хэшем в некоторых ситуациях) и начальные данные смарт-контракта (может быть пустым; может быть заменено его хэшем). Хэш кода и данных, содержащиеся в сообщении-конструкторе данные, должны совпадать с адресом η смарт-контракта, в противном случае сообщение отклоняется.

¹¹ Сообщения, несущие значения, с установленным флагом **bounce** не будут приняты неинициализированным аккаунты, а будут "отброшены" обратно.

После того как код и данные смарт-контракта инициализируются из тела сообщения-конструктора, оставшаяся часть сообщения-конструктора обрабатывается транзакцией (создающей транзакцией для смарт-контракта η) с использованием TVM аналогично обработке обычных входящих сообщений.

1.7.4. Начальный баланс смарт-контракта. Обратите внимание, что сообщение-конструктор обычно должно нести некоторое значение, которое будет переведено на баланс нового смарт-контракта; в противном случае новый смарт-контракт имел бы нулевой баланс и не смог бы оплатить хранение своего кода и данных в блокчейне. Минимальный баланс, необходимый для нового смарт-контракта, является линейной (точнее, аффинной) функцией от объема используемого хранилища. Коэффициенты этой функции могут зависеть от воркчейна; в частности, они выше в мастерчейне, чем в базовом воркчейне.

1.7.5. Создание смарт-контрактов внешними сообщениями-конструкторами. В некоторых случаях необходимо создать смарт-контракт с помощью сообщения-конструктора, которое не может нести никакого значения — например, с помощью сообщения-конструктора "из ниоткуда" (внешнего входящего сообщения). Тогда сначала нужно перевести достаточное количество средств на неинициализированный смарт-контракт, как объяснено в разделе [1.7.2](#), и только потом отправить сообщение-конструктор "из ниоткуда".

1.7.6. Пример: создание смарт-контракта криптовалютного кошелька. Примером вышеописанной ситуации является создание специальных приложений криптовалютных кошельков для пользователей, которым необходимо создать специальный смарт-контракт кошелька в блокчейне для хранения средств пользователя. Это можно сделать следующим образом:

- Приложение криптовалютного кошелька генерирует новую криптографическую пару ключей (публичный/приватный ключ) (как правило, для криптографии на эллиптических кривых Ed25519, поддерживаемой специальными примитивами TVM) для подписания будущих транзакций пользователя.
- Приложение криптовалютного кошелька знает код создаваемого смарт-контракта (который обычно одинаков для всех пользователей), а также данные, которые обычно включают в себя публичный ключ кошелька (или его хэш) и генерируются в самом начале. Хэш этой информации является адресом ζ создаваемого смарт-контракта кошелька.
- Приложение кошелька может отображать адрес ζ пользователя, и пользователь может начать получать средства на свой неинициализированный аккаунт ζ — например, купив некоторую криптовалюту на бирже или попросив друга перевести небольшую сумму.
- Приложение кошелька может проверить шардчейн, содержащий аккаунт ζ (в случае аккаунта базового воркчейна) или мастерчейн (в случае аккаунта мастерчейна), либо самостоятельно, либо используя блокчейн-explorer, и проверить баланс ζ .
- Если баланс достаточен, приложение кошелька может создать и подписать (с помощью приватного ключа пользователя) сообщение-конструктор ("из

ниоткуда") и отправить его для включения валидаторам или сборщикам для соответствующего блокчейна.

- Как только сообщение-конструктор включено в блок блокчейна и обработано транзакцией, смарт-контракт кошелька наконец создается.
- Когда пользователь хочет перевести средства другому пользователю или смарт-контракту η или хочет отправить сообщение, несущие значение к η , он использует свое приложение кошелька для создания сообщения m , которое он хочет, чтобы его смарт-контракт кошелька ζ отправил η , вложит m в специальное "сообщение из ниоткуда" m' с адресом назначения ζ , и подпишет m' своим приватным ключом. Некоторые положения против атак повторного воспроизведения должны быть сделаны, как объяснено в **2.2.1**.
- Смарт-контракт кошелька получает сообщение m' и проверяет правильность подписи с помощью публичного ключа, хранимого в его постоянных данных. Если подпись правильная, он извлекает вложенное сообщение m из m' и отправляет его в намеченное место назначения η с указанной суммой прикрепленных средств.
- Если пользователю не нужно немедленно начинать переводить средства, но он хочет просто получать некоторые средства, он может держать свой аккаунт неинициализированным столько, сколько захочет (при условии, что платежи за постоянное хранилище не приведут к исчерпанию его баланса), тем самым минимизируя профиль хранения и постоянные платежи за хранение учетной записи.
- Обратите внимание, что приложение кошелька может создавать для пользователя иллюзию, что средства хранятся в самом приложении, и предоставлять интерфейс для перевода средств или отправки произвольных сообщений "напрямую" с аккаунта пользователя ζ . На самом деле все эти операции будут выполняться смарт-контрактом кошелька пользователя, который фактически выступает прокси для таких запросов. Мы видим, что криптовалютный кошелек является простым примером смешанного приложения, имеющего часть в блокчейне (смарт-контракт кошелька, используемый как прокси для исходящих сообщений) и часть вне блокчейна (внешнее приложение кошелька, работающее на устройстве пользователя и сохраняющее приватный ключ учетной записи).

Конечно, это всего лишь один из способов работы с самыми простыми смарт-контрактами кошельков пользователя. Можно создать смарт-контракты кошельков с мультиподписью или создать общий кошелек с внутренними балансами, хранимыми внутри него для каждого из его индивидуальных пользователей и так далее.

1.7.7. Смарт-контракты могут быть созданы другими смарт-контрактами. Обратите внимание, что смарт-контракт может генерировать и отправлять сообщение-конструктор при обработке любой транзакции. Таким образом, смарт-контракты могут автоматически создавать новые смарт-контракты, если это необходимо, без какого-либо вмешательства человека.

1.7.8. Смарт-контракты могут быть созданы смарт-контрактами кошельков. С другой стороны, пользователь может скомпилировать код для своего нового смарт-контракта v , сгенерировать соответствующее сообщение-конструктор m , и использовать

приложение кошелька, чтобы заставить свой смарт-контракт кошелька ξ отправить сообщение m к v с достаточным количеством средств, тем самым создавая новый смарт-контракт v .

1.8. Модификация и удаление смарт-контрактов

Этот раздел объясняет, как могут быть изменены код и состояние смарт-контракта, и как и когда смарт-контракт может быть уничтожен.

1.8.1. Модификация данных смарт-контракта. Постоянные данные смарт-контракта обычно изменяются в результате выполнения кода смарт-контракта в TVM при обработке транзакций, вызванной входящим сообщением в смарт-контракт. Более конкретно, код смарт-контракта имеет доступ к старому постоянному хранилищу смарт-контракта через регистр TVM `c4` и может изменять постоянное хранилище, сохраняя другое значение в `c4` перед нормальным завершением.

Обычно других способов изменения данных существующего смарт-контракта нет. Если код смарт-контракта не предусматривает никаких способов изменения постоянных данных (например, если это простой смарт-контракт кошелька, описанный в разделе [1.7.6](#), который инициализирует постоянные данные публичным ключом пользователя и не намерен его когда-либо менять), то он будет фактически не изменяемым — до тех пор, пока не будет изменен сам код смарт-контракта.

1.8.2. Изменение кода смарт-контракта. Точно так же, код существующего смарт-контракта может быть изменен только в том случае, если в текущем коде предусмотрены такие изменения. Код изменяется путем вызова примитива TVM `SETCODE`, который устанавливает корень кода для текущего смарт-контракта из верхнего значения в стеке TVM. Изменение применяется только после нормального завершения текущей транзакции.

Обычно, если разработчик смарт-контракта хочет иметь возможность обновлять его код в будущем, он предусматривает специальный "метод обновления кода" в оригинальном коде смарт-контракта, который вызывает `SETCODE` в ответ на определенные входящие сообщения "обновления кода", используя новый код, отправленный в сообщении, в качестве аргумента для `SETCODE`. Должны быть предусмотрены некоторые меры для защиты смарт-контракта от несанкционированной замены кода; в противном случае можно потерять контроль над смарт-контрактом и средствами на его балансе. Например, сообщения об обновлении кода могут приниматься только с доверенного адреса источника или защищаться с помощью действительной криптографической подписи и правильного номера последовательности.

1.8.3. Хранение кода или данных смарт-контракта вне блокчейна. Код или данные смарт-контракта могут храниться вне блокчейна и представляться только их хэшами. В таких случаях могут обрабатываться только пустые входящие сообщения, а также сообщения, содержащие правильную копию кода смарт-контракта (или его части, относящейся к обработке конкретного сообщения) и его данные внутри специальных

полей. Пример такой ситуации приведен для неинициализированных смарт-контрактов и сообщений-конструкторов, описанных в разделе [1.7](#).

1.8.4. Использование библиотек кода. Некоторые смарт-контракты могут использовать один и тот же код, но разные данные. Одним из примеров этого являются смарт-контракты кошельков (см. [1.7.6](#)), которые, вероятно, будут использовать один и тот же код (во всех кошельках, созданных одним и тем же программным обеспечением), но с разными данными (поскольку каждый кошелек должен использовать свою пару криптографических ключей). В этом случае код для всех смарт-контрактов кошельков лучше всего помещать разработчиком в общую *библиотеку*; эта библиотека будет находиться в мастерчайне и ссылаться на ее хэш с использованием специальной "внешнего ссылки на ячейку библиотеки" в качестве корня кода каждого смарт-контракта кошелька (или как поддерево внутри этого кода).

Обратите внимание, что даже если код библиотеки станет недоступным, например, потому что ее разработчик перестанет платить за ее хранение в мастерчайне, все равно можно будет использовать смарт-контракты, ссылающиеся на эту библиотеку, либо снова добавив библиотеку в мастерчайн, либо включив ее соответствующие части внутрь сообщения, отправленного смарт-контракту. Этот механизм разрешения внешних ссылок обсуждается более подробно позже в разделе [4.4.3](#).

1.8.5. Уничтожение смарт-контрактов. Обратите внимание, что смарт-контракт не может быть действительно уничтожен, пока его баланс не станет нулевым или отрицательным. Это может произойти в результате сбора платежей за постоянное хранение или после отправки исходящего сообщения, несущего значение, которое переводит почти все его предыдущие средства.

Например, пользователь может решить перевести все оставшиеся средства с кошелька на другой кошелек или смарт-контракт. Это может быть полезно, например, если пользователь хочет обновить кошелек, но смарт-контракт кошелька не предусматривает никаких будущих обновлений; тогда можно просто создать новый кошелек и перевести все средства на него.

1.8.6. Замороженные аккаунты. Когда баланс аккаунта становится неположительным после транзакции или меньше определенного минимума, зависящего от воркчайна, аккаунт *замораживается* путем замены всего его кода и данных на один 32-байтовый хэш. Этот хэш сохраняется некоторое время (например, пару месяцев), чтобы предотвратить воссоздание смарт-контракта его оригинальной создающей транзакцией (которая все еще имеет правильный хэш, равный адресу аккаунта) и позволить ее владельцу восстановить аккаунт, переведя некоторые средства и отправив сообщение, содержащее код и данные аккаунта, для восстановления в блокчайне. В этом отношении замороженные аккаунты похожи на неинициализированные аккаунты; однако хэш правильного кода и данных для замороженного аккаунта не обязательно равен адресу аккаунта, а хранится отдельно.

Обратите внимание, что замороженные аккаунты могут иметь отрицательный баланс, указывающий на то, что наступили сроки платежей за постоянное хранение.

Аккаунт не может быть разморожен до тех пор, пока его баланс не станет положительным и больше предписанного минимального значения.

2 Пересылка сообщений и гарантии доставки

Эта глава о пересылке сообщений внутри TON Blockchain, включая протоколы Hypercube Routing (HR) и Instant Hypercube Routing (IHR). Также описываются условия, необходимые для реализации гарантий доставки сообщений и гарантии упорядочивания FIFO.

2.1 Адреса сообщений и вычисление следующего перехода

Этот раздел объясняет вычисление транзитных адресов и адресов следующего перехода по варианту алгоритма гиперкубической маршрутизации, используемой в TON Blockchain. Сам протокол гиперкубической маршрутизации, который использует концепции и алгоритм вычисления адресов следующего перехода, введенные в этом разделе, представлен в следующем разделе.

2.1.1. Адреса аккаунтов. *Адрес отправителя и адрес получателя* всегда присутствуют в любом сообщении. Обычно это (полные) *адреса аккаунтов*. Полный адрес аккаунта состоит из *workchain_id* (подписанного 32-битного целого little-endian числа, определяющего воркчейн), за которым следует (обычно) 256-битный *внутренний адрес* или *идентификатор аккаунта account_id* (который также может интерпретироваться как неподписанное целое little-endian число), определяющий аккаунт в выбранном воркчейне.

Разные воркчейны могут использовать идентификаторы аккаунтов, которые короче или длиннее «стандартных» 256 бит, используемых в мастерчейне (*workchain_id = -1*) и в базовом воркчейне (*workchain_id = 0*). С этой целью состояние мастерчайна содержит список всех воркчайнов, определенных на данный момент, вместе с их длинами идентификаторов аккаунтов. Важное ограничение состоит в том, что длина *account_id* для любого воркчайна должна быть не менее 64 бит.

В дальнейшем для простоты мы часто рассматриваем только случай 256-битных адресов аккаунтов. Только первые 64 бита *account_id* имеют значение для целей маршрутизации сообщений и разделения шардчайна.

2.1.2. Адреса отправителя и получателя сообщения. Любое сообщение имеет как *адрес отправителя*, так и *адрес получателя*. Адрес отправителя — это адрес аккаунта (смарт-контракта), который создал сообщение при обработке какой-либо транзакции; адрес отправителя нельзя изменить или задать произвольно, и смарт-контракты сильно полагаются на это свойство. Напротив, при создании сообщения может быть выбран любой корректно сформированный адрес получателя; после этого адрес получателя изменить нельзя.

2.1.3. Внешние сообщения без адреса отправителя или получателя. Некоторые сообщения могут не иметь адреса отправителя или получателя (хотя, по крайней мере, один из них должен быть), как указано специальными флагами в заголовке сообщения. Такие сообщения называются *внешними сообщениями*, предназначенными для взаимодействия TON Blockchain с внешним миром — пользователями и их криптокошельками, внецепочечными и смешанными приложениями и сервисами, другими блокчейнами и т. д.

Внешние сообщения никогда не маршрутизируются внутри TON Blockchain. Вместо этого "сообщения из ниоткуда" (т. е. без адреса отправителя) включаются напрямую в *InMsgDescr* целевого блока шардчайна (при соблюдении некоторых условий) и обрабатываются транзакцией в этом самом блоке. Аналогично, "сообщения в никуда" (т. е. без адреса получателя в TON Blockchain), также известные как *log-сообщения*, присутствуют только в блоке, содержащем транзакцию, которая сгенерировала такое сообщение.^[12]

Таким образом, внешние сообщения почти не имеют значения для обсуждения маршрутизации сообщений и гарантий доставки сообщений. На самом деле, гарантии доставки сообщений для исходящих внешних сообщений тривиальны (в лучшем случае, сообщение должно быть включено в часть *LogMsg* блока), а для входящих внешних сообщений их вообще нет, так как валидаторы блока шардчайна могут по своему усмотрению включать или игнорировать предложенные входящие внешние сообщения (например, в зависимости от предлагаемой платы за обработку).^[13]

В дальнейшем мы сосредоточимся на "обычных" или "внутренних" сообщениях, которые имеют как адрес отправителя, так и адрес получателя.

2.1.4. Транзитные и адреса следующего перехода. Когда сообщение необходимо маршрутизировать через промежуточные шардчайны, прежде чем оно достигнет предполагаемого места назначения, ему присваивается *транзитный адрес и адрес следующего перехода* в дополнение к (неизменяемым) адресам отправителя и получателя. Когда копия сообщения находится в транзитном шардчайне в ожидании пересылки на следующий переход, *транзитный адрес* — это его промежуточный адрес, находящийся в транзитном шардчайне, как если бы он принадлежал специальному смарт-контракту для передачи сообщений, чья единственная задача — передать неизмененное сообщение в следующий шардчайн на маршруте. Адрес следующего перехода — это адрес в соседнем шардчайне (или, в некоторых редких случаях, в том же шардчайне), в который нужно переслать сообщение. После пересылки сообщения адрес следующего перехода обычно становится транзитным адресом копии сообщения, включенной в следующий шардчайн.

¹² "Сообщения в никуда" могут иметь специальные поля в своем теле, указывающие их место назначения за пределами TON Blockchain — например, аккаунт в другом блокчайне, или IP-адрес и порт, которые могут быть интерпретированы соответствующим сторонним программным обеспечением. Такие поля игнорируются TON Blockchain.

¹³ Проблема обхода возможной цензуры валидаторами — что может произойти, например, если все валидаторы согласятся не включать внешние сообщения, отправленные на аккаунты из некоторого набора заблокированных аккаунтов — решается отдельно. Основная идея заключается в том, что валидаторы могут быть вынуждены обещать включать сообщение с известным хэшем в будущий блок, не зная ничего о личности отправителя или получателя; они должны будут выполнить это обещание впоследствии, когда сообщение с заранее согласованным хэшем будет представлено.

Сразу после создания исходящего сообщения в шардчейне (или в мастерчейне) его транзитный адрес устанавливается равным адресу отправителя.¹⁴

2.1.5. Вычисление адреса следующего перехода для гиперкубической маршрутизации. TON Blockchain использует вариант гиперкубической маршрутизации. Это означает, что адрес следующего перехода вычисляется из транзитного адреса (первоначально равного адресу отправителя) следующим образом:

1. Компоненты 32-битного $workchain_id$ (подписанного little-endian числа) как транзитного адреса, так и адреса назначения делятся на группы по n_1 бит (в настоящее время $n_1 = 32$), и они сканируются слева (т. е. от наиболее значимых битов) направо. Если одна из групп в транзитном адресе отличается от соответствующей группы в адресе назначения, то значение этой группы в транзитном адресе заменяется значением в адресе назначения для вычисления адреса следующего перехода.
2. Если части $workchain_id$ транзитного и целевого адресов совпадают, то аналогичный процесс применяется к частям $account_id$ адресов: части $account_id$, или, точнее, их первые (наиболее значимые) 64 бита делятся на группы по n_2 бит (в настоящее время $n_2 = 4$ битовые группы используются, соответствующие шестнадцатеричным цифрам адреса), начиная с наиболее значимого бита и сравниваются, начиная с левого. Первая группа, которая отличается, заменяется в транзитном адресе своим значением в адресе назначения для вычисления адреса следующего перехода.
3. Если первые 64 бита частей $account_id$ транзитного и целевого адресов также совпадают, то аккаунт назначения принадлежит текущему шардчейну и сообщение не должно быть переслано за пределы текущего шардчейна. Вместо этого он должен быть обработан транзакцией внутри него.

2.1.6. Обозначение для адреса следующего перехода. Мы обозначаем

$$\text{NEXTHOP}(\xi, \eta) \tag{13}$$

адрес следующего перехода, вычисленный для текущего (адрес отправителя или транзитного адреса) ξ и адреса назначения η .

2.1.7. Поддержка anycast-адресов. "Большие" смарт-контракты, которые могут иметь отдельные экземпляры в разных шардчейнах, могут быть достигнуты с использованием anycast-адресов назначения. Эти адреса поддерживаются следующим образом.

Anycast-адрес (η, d) состоит из обычного адреса η вместе с его "глубиной разбиения" $d \leq 31$. Идея заключается в том, что сообщение может быть доставлено на любой адрес, отличающийся от η только в первых d битах внутренней части адреса (т. е. не включая идентификатор воркчайна, который должен совпадать точно). Это достигается следующим образом:

¹⁴ Однако внутренняя маршрутизация, описанная в разделе 2.1.11, применяется немедленно после этого, что может дополнительно изменить транзитный адрес.

- Эффективный адрес назначения $\tilde{\eta}$ вычисляется из (η, d) путем замены первых d битов внутренней части адреса η соответствующими битами, взятыми из адреса отправителя ξ .
- Все вычисления $\text{NEXTHOP}(v, \eta)$ заменяются на $\text{NEXTHOP}(v, \tilde{\eta})$ для $v = \xi$ и для всех других промежуточных адресов v . Таким образом, Hypercube Routing или Instant Hypercube Routing в итоге доставят сообщение в шардчейн, содержащий $\tilde{\eta}$.
- Когда сообщение обрабатывается в своем целевом шардчейне (тот, который содержит адрес $\tilde{\eta}$), оно может быть обработано аккаунтом η' того же шардчейна, отличающейся от η и $\tilde{\eta}$ только в первых d битах внутренней части адреса. Точнее говоря, если общий префикс адреса шардчейна — это s , так что только внутренние адреса, начинающиеся с двоичной строки s , относятся к целевому шардчейну, то η' вычисляется из η путем замены первых $\min(d, |s|)$ бит внутренней части адреса η соответствующими битами s .

Таким образом, в дальнейших обсуждениях мы игнорируем существование anycast-адресов и дополнительную обработку, которую они требуют.

2.1.8. Оптимальность алгоритма вычисления адреса следующего перехода по Хэммингу. Обратите внимание, что специфический алгоритм гиперкубической маршрутизации вычисления адреса следующего перехода, объясненный в разделе 2.1.5, может быть потенциально заменен другим алгоритмом, при условии, что он удовлетворяет определенным свойствам. Одним из таких свойств является *оптимальность по Хэммингу*, что означает, что расстояние Хэмминга (L_1) от ξ до η равно сумме расстояний Хэмминга от ξ до $\text{NEXTHOP}(\xi, \eta)$ и от $\text{NEXTHOP}(\xi, \eta)$ до η :

$$\|\xi - \eta\|_1 = \|\xi - \text{NEXTHOP}(\xi, \eta)\|_1 + \|\text{NEXTHOP}(\xi, \eta) - \eta\|_1 \quad (14)$$

Здесь $\|\xi - \eta\|_1$ — это *расстояние Хэмминга* между ξ и η , равное количеству битов, в которых ξ и η различаются:¹⁵

$$\|\xi - \eta\|_1 = \sum_i |\xi_i - \eta_i| \quad (15)$$

Обратите внимание, что в общем случае можно ожидать только неравенство (14), следующее из неравенства треугольника для метрики L_1 . Оптимальность по Хэммингу в сущности означает, что $\text{NEXTHOP}(\xi, \eta)$ лежит на одном из (Хэмминговых) кратчайших путей от ξ до η . Это также можно выразить, сказав, что $v = \text{NEXTHOP}(\xi, \eta)$ всегда получается из ξ путем изменения значений битов в некоторых позициях на их значения в η : для любой битовой позиции i мы имеем $v_i = \xi_i$ или $v_i = \eta_i$.¹⁶

¹⁵ Когда адреса имеют разные длины (например, потому что они принадлежат разным воркчейнам), следует учитывать только первые 96 битов адресов в приведенной выше формуле.

¹⁶ Вместо оптимальности по Хэммингу можно рассмотреть эквивалентное свойство оптимальности Kademlia, записанное для расстояния Kademlia (или взвешенного L) как $\|\xi - \eta\|_k := \sum_i 2^{-i} |\xi_i - \eta_i|$ вместо расстояния Хэмминга.

2.1.9. Непрерывность NEXTHOP. Еще одно важное свойство NEXTHOP — это его *непрерывность*, что означает, что $\text{NEXTHOP}(\xi, \eta) = \xi$ возможно только если $\xi = \eta$. Другими словами, если мы еще не прибыли в η , следующий шаг не может совпадать с нашей текущей позицией.

Это свойство означает, что путь от ξ до η , то есть последовательность промежуточных адресов $\xi^{(0)} := \xi$, $\xi^{(n)} := \text{NEXTHOP}(\xi^{(n-1)}, \eta)$ будет постепенно стабилизироваться на η : для некоторого $N \geq 0$, мы имеем $\xi^{(n)} = \eta$ для всех $n \geq N$. Действительно, всегда можно взять $N := \|\xi - \eta\|_1$.

2.1.10. Выпуклость пути HR относительно шардирования. Последовательность свойств оптимальности по Хэммингу (14) — это то, что мы называем *выпуклостью* пути от ξ до η относительно шардирования. А именно, если $\xi^{(0)} := \xi$, $\xi^{(n)} := \text{NEXTHOP}(\xi^{(n-1)}, \eta)$ — это вычисленный путь от ξ до η , и N — это первый индекс, такой что $\xi^{(N)} = \eta$, и S — это шард какого-либо воркчайна в любой конфигурации шардов, то индексы i , при которых $\xi^{(i)}$ находится в шарде S , составляют под интервал в $[0, N]$. Другими словами, если целые числа $0 \leq i \leq j \leq k \leq N$ таковы, что $\xi^{(i)}, \xi^{(k)} \in S$, то и $\xi^{(j)} \in S$.

Это свойство выпуклости важно для некоторых доказательств, связанных с пересылкой сообщений в условиях динамического шардирования.

2.1.11. Внутренняя маршрутизация. Обратите внимание, что адрес следующего перехода, вычисленный по правилам, определенным в разделе 2.1.5, может принадлежать тому же шардчайну, что и текущий (т. е. содержащий транзитный адрес). В этом случае "внутренняя маршрутизация" происходит немедленно, транзитный адрес заменяется значением вычисленного адреса следующего перехода, и шаг вычисления адреса следующего перехода повторяется до тех пор, пока не будет получен адрес следующего перехода, лежащий вне текущего шардчайна. Сообщение затем сохраняется в выходной очереди транзита в соответствии с его вычисленным адресом следующего перехода с его последним вычисленным транзитным адресом в качестве "промежуточного владельца" транзитного сообщения. Если текущий шардчайн разделяется на два шардчайна до того, как сообщение будет переслано дальше, это будет шардчайн, содержащий промежуточного владельца, который наследует это транзитное сообщение.

Альтернативно можно продолжать вычислять адреса следующего перехода, пока не выяснится, что адрес назначения уже принадлежит текущему шардчайну. В этом случае сообщение будет обработано (транзакцией) внутри этого шардчайна, а не переслано дальше.

2.1.12. Соседние шардчайны. Два шарда в конфигурации шардов или соответствующие два шардчайна считаются *соседями* или *соседними шардчайнами*, если один из них содержит адрес следующего перехода хотя бы для одной комбинации допустимых адресов отправителя и получателя, в то время как другой содержит транзитный адрес для той же комбинации. Другими словами, два шардчайна являются соседями, если сообщение может быть переслано напрямую из одного из них в другой через Hyperscibe Routing.

Мастерчайн также включен в это определение, как если бы он был единственным шардчайном воркчайна с *workchain_id* = -1. В этом отношении он является соседом всех других шардчайнов.

2.1.13. Любой шард является соседом самому себе. Обратите внимание, что шардчайн всегда считается соседом самому себе. Это может показаться излишним, поскольку мы всегда повторяем вычисление адреса следующего перехода, описанное в разделе [2.1.5](#) до тех пор, пока не получим адрес следующего перехода вне текущего шардчайна (см. [2.1.11](#)). Однако есть как минимум две причины для такого порядка:

- Некоторые сообщения имеют адреса отправителя и получателя внутри одного и того же шардчайна, по крайней мере, когда сообщение создается. Однако, если такое сообщение не обрабатывается немедленно в том же блоке, где оно было создано, оно должно быть добавлено в очередь исходящих сообщений своего шардчайна и импортировано как входящее сообщение (с записью в *InMsgDescr*) в одном из последующих блоков того же шардчайна.¹⁷
- В качестве альтернативы адрес следующего перехода может первоначально находиться в каком-то другом шардчайне, который позже объединяется с текущим шардчайном, так что следующий шаг становится внутри того же шардчайна. Тогда сообщение должно быть импортировано из очереди исходящих сообщений объединенного шардчайна и переслано или обработано в соответствии с его адресом следующего перехода, даже если они теперь находятся внутри одного шардчайна.

2.1.14. Hypercube Routing и ISP. В конечном счете здесь применяется Infinite Sharding Paradigm (ISP): шардчайн должен рассматриваться как временный союз аккаунтчайнов, объединенных исключительно для минимизации накладных расходов на генерацию и передачу блоков.

Пересылка сообщения проходит через несколько промежуточных аккаунтчайнов, некоторые из которых могут находиться в одном и том же шарде. В этом случае, как только сообщение достигает аккаунтчайна, находящегося в этом шарде, оно немедленно ("внутренне") маршрутизируется внутри этого шардчайна до тех пор, пока не будет достигнуто последний аккаунтчайн, находящийся в том же шарде (см. [2.1.11](#)). Затем сообщение ставится в очередь вывода этого последнего аккаунтчайна.¹⁸

¹⁷ Обратите внимание, что вычисления следующего перехода и внутренней маршрутизации все еще применяются к таким сообщениям, поскольку текущий шардчайн может быть разделен до того, как сообщение будет обработано. В этом случае новый подшардчайн, содержащий адрес назначения, унаследует сообщение.

¹⁸ Мы можем определить (виртуальную) очередь вывода аккаунта (аккаунтчайна) как подмножество *OutMsgQueue* шарда, содержащего этот аккаунт, которое состоит из сообщений с транзитными адресами, равными адресу аккаунта.

2.1.15. Представление транзитных адресов и адресов следующего перехода. Обратите внимание, что транзитный адрес и адрес следующего перехода отличаются от адреса отправителя только в *workchain_id* и в первых (наиболее значимых) 64 битах адреса аккаунта. Поэтому они могут быть представлены в виде 96-битных строк. Более того, их *workchain_id* обычно совпадает с *workchain_id* либо адреса отправителя, либо адреса получателя; пара битов может использоваться для указания этой ситуации, что еще больше сокращает пространство, необходимое для представления транзитных и следующих адресов.

Фактически, необходимое хранилище может быть сокращено еще больше, если учесть, что специфический алгоритм гиперкубической маршрутизации, описанный в разделе [2.1.5](#), всегда генерирует промежуточные (т. е. транзитные и следующего перехода) адреса, которые совпадают с адресом назначения в их первых k битах и с адресом отправителя в их оставшихся битах. Поэтому можно использовать только значения $0 \leq k_{tr}, k_{nh} \leq 96$, чтобы полностью указать транзитные и адреса следующего перехода. Можно также заметить, что $k' := k_{nh}$ оказывается фиксированной функцией от $k := k_{tr}$ (например, $k' = k + n_2 = k + 4$ для $k \geq 32$), и поэтому включить только одно 7-битное значение k в сериализацию.

Такие оптимизации имеют очевидный недостаток, так как они слишком сильно зависят от конкретного используемого алгоритма маршрутизации, который может быть изменен в будущем, поэтому они используются в разделе [3.1.15](#) с условием указания более общих промежуточных адресов, если это необходимо.

2.1.16. Оболочки сообщений. Транзитные и адреса следующего перехода пересылаемого сообщения не включены в само сообщение, а хранятся в специальной *оболочке сообщения*, которая является ячейкой (или срезом ячейки), содержащей транзитные и адреса следующего перехода с вышеупомянутыми оптимизациями, некоторую другую информацию, относящуюся к пересылке и обработке, и ссылку на ячейку, содержащую неизмененное оригинальное сообщение. Таким образом, сообщение можно легко "извлечь" из его оригинальной оболочки (например, той, которая присутствует в *InMsgDescr*) и поместить в другую оболочку (например, перед включением в *OutMsgQueue*).

В представлении блока в виде дерева или, скорее, DAG ячеек, две разные оболочки будут содержать ссылки на общую ячейку с оригинальным сообщением. Если сообщение велико, это расположение позволяет избежать необходимости хранить более одной копии сообщения в блоке.

2.2 Протокол Hypercube Routing

В этом разделе изложены детали протокола гиперкубической маршрутизации, используемого в TON Blockchain для обеспечения гарантированной доставки сообщений между смарт-контрактами, находящимися в произвольных шардчайнах. Для целей этого

документа мы будем ссыльаться на вариант гиперкубической маршрутизации, используемой в TON Blockchain как Hypercube Routing (HR).

2.2.1. Уникальность сообщений. Прежде чем продолжить, давайте отметим, что любое (внутреннее) сообщение *的独特*. Напомним, что сообщение содержит полный адрес отправителя вместе с его логическим временем создания, и все исходящие сообщения, созданные одним и тем же смарт-контрактом, имеют строго возрастающие логические времена создания (см. [1.4.6](#)); поэтому комбинация полного адреса отправителя и логического времени создания уникально определяет сообщение. Поскольку мы предполагаем, что выбранная хэш-функция SHA256 устойчива к коллизиям, *сообщение* *的独特* *определяется своим хэшем*, поэтому мы можем идентифицировать два сообщения, если знаем, что их хэши совпадают.

Это не распространяется на внешние сообщения "из ниоткуда", у которых нет адресов отправителя. Особое внимание должно быть уделено предотвращению атак повторного воспроизведения, связанных с такими сообщениями, особенно архитекторами смарт-контрактов пользовательских кошельков. Одним из возможных решений является включение номера последовательности в тело таких сообщений и ведение учета внешних сообщений, уже обработанных внутри постоянных данных смарт-контракта, отказываясь обрабатывать внешнее сообщение, если его номер последовательности отличается от этого учета.

2.2.2. Идентификация сообщений с одинаковыми хэшами. TON Blockchain предполагает, что два сообщения с одинаковыми хэшами совпадают, и рассматривает одно из них как избыточную копию другого. Как объяснено выше в [2.2.1](#), это не приводит к неожиданным последствиям для внутренних сообщений. Однако, если отправить два совпадающих "сообщения из ниоткуда" в смарт-контракт, может случиться так, что будет доставлено только одно из них или оба. Если их действие не предполагается идемпотентным (т. е. если обработка сообщения дважды имеет другой эффект, чем его обработка один раз), необходимо предусмотреть различие между двумя сообщениями, например, включив в них номер последовательности.

В частности, *InMsgDescr* и *OutMsgDescr* используют хэш сообщения (без оболочки) в качестве ключа, предполагая, что разные сообщения имеют разные хэши. Таким образом, можно проследить путь и судьбу сообщения через разные шардчайны, просматривая хэш сообщения в *InMsgDescr* и *OutMsgDescr* разных блоков.

2.2.3. Структура *OutMsgQueue*. Напомним, что исходящие сообщения, как те, которые созданы внутри шардчайна, так и транзитные сообщения, ранее импортированные из соседнего шардчайна для пересылки в шардчайн следующего перехода, накапливаются в *OutMsgQueue*, которая является частью *состояния* шардчайна (см. [1.2.7](#)). В отличие от *InMsgDescr* и *OutMsgDescr*, ключ в *OutMsgQueue* — это не хэш сообщения, а его адрес следующего перехода или, по крайней мере, его первые 96 бит, соединенные с хэшем сообщения.

Кроме того, *OutMsgQueue* — это не просто словарь (hashmap), сопоставляющий свои ключи с сообщениями (с оболочкой). Скорее, это *минимально расширенный словарь с точки зрения логического времени создания*, что означает, что каждый узел дерева

Патриция, представляющего *OutMsgQueue*, имеет дополнительное значение (в данном случае беззнаковое 64-битное целое число), и это дополнительное значение в каждом узле-развилке устанавливается равным минимуму из дополнительных значений его дочерних узлов. Дополнительное значение листа равно логическому времени создания сообщения, содержащегося в этом листе; его не обязательно хранить явно.

2.2.4. Инспектирование *OutMsgQueue* соседа. Такая структура для *OutMsgQueue* позволяет валидаторам соседнего шардчайна проверять его, чтобы найти его часть (поддерево Патриции), относящуюся к ним (т. е. состоящую из сообщений с адресом следующего перехода, принадлежащего соседнему шарду или имеющего адрес следующего перехода с данным двоичным префиксом), а также быстро вычислить "старейшее" (т. е. с минимальным логическим временем создания) сообщение в этой части.

Кроме того, валидаторам шардчайна даже не нужно отслеживать общее состояние всех их соседних шардчайнов — им нужно только хранить и обновлять копию своего *OutMsgQueue* или даже его поддерева, относящегося к ним.

2.2.5. Монотонность логического времени: импорт самого старого сообщения от соседей. Первое фундаментальное локальное условие пересылки сообщений, называемое *условие монотонности (логического времени) (импорта сообщения)*, можно резюмировать следующим образом:

При импорте сообщений в *InMsgDescr* блока шардчайна из *OutMsgQueue* его соседних шардчайнов валидаторы должны импортировать сообщения в порядке возрастания их логического времени; в случае равенства импортируется сообщение с меньшим хэшем.

Более точно, каждый блок шардчайна содержит хэш блока мастерчайна (предполагается "самым последним" блоком мастерчайна на момент создания блока шардчайна), который, в свою очередь, содержит хэши самых последних блоков шардчайнов. Таким образом, каждый блок шардчайна косвенно "знает" о самом последнем состоянии всех других шардчайнов, особенно его соседних шардчайнов, включая их *OutMsgQueue*.¹⁹

Теперь альтернативная эквивалентная формулировка условия монотонности заключается в следующем:

Если сообщение импортируется в *InMsgDescr* нового блока, его логическое время создания не может быть больше, чем у любого сообщения, оставшегося не импортированным в *OutMsgQueue* самого последнего состояния любого из соседних шардчайнов.

Именно эта форма условия монотонности появляется в локальных условиях консенсуса блоков TON Blockchain и обеспечивается валидаторами.

¹⁹ В частности, если хэш недавнего блока соседнего шардчайна еще не отражен в последнем блоке мастерчайна, его изменения в *OutMsgQueue* не должны приниматься в расчет.

2.2.6. Свидетели нарушений условия монотонности логического времени импорта сообщений. Обратите внимание, что если это условие не выполняется, может быть создано небольшое доказательство Меркла, свидетельствующее о его нарушении. Такое доказательство будет содержать:

- Путь в *OutMsgQueue* соседа от корня к определенному сообщению m с небольшим логическим временем создания.
- Путь в *InMsgDescr* рассматриваемого блока, показывающий, что ключ, равный $\text{HASH}(m)$, отсутствует в *InMsgDescr* (т. е. что m не было включено в текущий блок).
- Доказательство того, что m не было включено в предыдущий блок того же шардчайна, используя информацию заголовка блока, содержащую наименьшее и наибольшее логическое время всех сообщений, импортированных в блок (см. [2.3.4](#) — [2.3.7](#) для получения дополнительной информации).
- Путь в *InMsgDescr* к другому включенному сообщению m' , такому что либо $\text{LT}(m') > \text{LT}(m)$, либо $\text{LT}(m') = \text{LT}(m)$ и $\text{HASH}(m') > \text{HASH}(m)$.

2.2.7. Удаление сообщения из *OutMsgQueue*. Сообщение должно быть удалено из *OutMsgQueue* рано или поздно; в противном случае, используемое *OutMsgQueue* хранилище будет расти до бесконечности. С этой целью вводятся несколько «правил сборки мусора».

Они позволяют удалить сообщение из *OutMsgQueue* во время оценки блока только в том случае, если в *InMsgDescr* этого блока присутствует явная специальная "запись о доставке". Эта запись содержит либо ссылку на соседний блок шардчайна, который включил сообщение в свой *InMsgDescr* (хэш блока достаточен, но сопутствующий материал для блока может содержать соответствующее доказательство Меркла), либо доказательство Меркла того факта, что сообщение было доставлено до конечного назначения через Instant Hypercube Routing.

2.2.8. Гарантируемая доставка сообщений через Hypercube Routing. Таким образом, сообщение не может быть удалено из очереди исходящих сообщений, если оно не было переслано в шардчайн следующего перехода или доставлено до конечного назначения (см. [2.2.7](#)). Между тем, условие монотонности импорта сообщений (см. [2.2.5](#)) гарантирует, что любое сообщение рано или поздно будет переслано в следующий шардчайн, принимая во внимание другие условия, которые требуют от валидаторов использовать как минимум половину пространства блока или лимитов газа для импорта входящих внутренних сообщений (иначе валидаторы могут выбрать создание пустых блоков или импорт только внешних сообщений, даже при наличии непустых очередей исходящих сообщений у их соседей).

2.2.9. Порядок обработки сообщений. Когда несколько импортированных сообщений обрабатываются транзакциями внутри блока, условия порядка обработки сообщений гарантируют, что старые сообщения обрабатываются первыми. Точнее, если блок содержит две транзакции t и t' одного и того же аккаунта, которые обрабатывают входящие сообщения m и m' соответственно, и $\text{LT}(m) < \text{LT}(m')$, то мы должны иметь $\text{LT}(t) < \text{LT}(t')$.

2.2.10. Гарантии FIFO Hypercube Routing. Условия порядка обработки сообщений (см. [2.2.9](#)), вместе с условиями монотонности импорта сообщений (см. [2.2.5](#)), подразумевают *гарантии FIFO для Hypercube Routing*. А именно, если смарт-контракт ζ генерирует два сообщения m и m' с одним и тем же адресом назначения η , и m' создано позже, чем m (что означает, что $m < m'$, следовательно $LT(m) < LT(m')$), тогда m будет обработано η перед m' . Это так потому, что оба сообщения будут следовать одним и тем же шагам маршрутизации на пути от ζ до η (алгоритм Hypercube Routing, описанный в разделе [2.1.5](#), является детерминированным), и во всех очередях исходящих и описаниях входящих сообщений m' будет появляться "после" m .²⁰

Если сообщение m' может быть доставлено в B через Instant Hypercube Routing, это уже необязательно будет верно. Поэтому простой способ обеспечить дисциплину доставки сообщений FIFO между парой смарт-контрактов состоит в установке специального бита в заголовке сообщения, предотвращающего его доставку через IHR.

2.2.11. Гарантии уникальности доставки Hypercube Routing. Обратите внимание, что условия монотонности импорта сообщений также подразумевают *уникальность* доставки любого сообщения через Hypercube Routing, т. е. оно не может быть импортировано и обработано смарт-контрактом назначения более одного раза. Мы увидим позже в разделе [2.3](#), что обеспечение уникальности доставки при активных Hypercube Routing и Instant Hypercube Routing более сложно.

2.2.12. Обзор Hypercube Routing. Позвольте нам обобщить все шаги маршрутизации, выполняемые для доставки внутреннего сообщения m , созданного исходным аккаунтом ζ_0 , в аккаунт назначения η . Мы обозначаем $\zeta_{k+1} := \text{NEXTHOP}(\zeta_k, \eta)$, $k = 0, 1, 2, \dots$ промежуточные адреса, предписанные HR для пересылки сообщения m до его конечного назначения η . Пусть S_k будет шардом, содержащим ζ_k .

²⁰ Это утверждение не так тривиально, как кажется на первый взгляд, потому что некоторые участвующие шардчейны могут разделяться или сливатся во время маршрутизации. Корректное доказательство может быть получено путем принятия перспективы ISP к HR, как объяснено в [2.1.14](#) и наблюдения, что m' всегда будет находиться позади m , либо в терминах промежуточного достигнутого аккаунтчайна, либо, если они оказываются в том же аккаунтчайне, в терминах логического времени создания.

Ключевое наблюдение заключается в том, что «в любой данный момент времени» (логически; более точное описание будет «в общем состоянии, полученном после обработки любого причинно-замкнутого подмножества блоков F »), промежуточные аккаунтчайны, принадлежащие тому же шарду, являются смежными на пути от ζ до η (т. е. не могут иметь аккаунтчайны, принадлежащие какому-либо другому шарду, между ними). Это свойство "выпуклости" (см. [2.1.10](#)) алгоритма Hypercube Routing, описанного в [2.1.5](#).

- [Birth] (Рождение) — Сообщение с адресом назначения η создается транзакцией t , принадлежащей аккаунту ζ_0 , находящейся в каком-либо шардчайне S_0 . Логическое время создания $LT(m)$ фиксируется в этот момент и включается в сообщение m .
- [ImmediateProcessing?] (Немедленная обработка?) — Если адрес назначения η находится в том же шардчайне S_0 , сообщение может быть обработано в том же блоке, в котором оно было создано. В этом случае m включается в $OutMsgDescr$ с флагом, указывающим, что оно было обработано в этом самом блоке и не требует дальнейшей пересылки. Другая копия m включается в $InMsgDescr$ вместе с обычными данными, описывающими обработку входящих сообщений. (Обратите внимание, что m не включается в $OutMsgQueue S_0$.)
- [InitialInternalRouting] (Первоначальная внутренняя маршрутизация) — Если сообщение m имеет адрес назначения вне S_0 или не обрабатывается в том же блоке, где оно было сгенерировано, применяется процедура внутренней маршрутизации, описанная в разделе 2.1.11, до тех пор, пока не будет найден индекс k , такой что ζ_k находится в S_0 , но $\zeta_{k+1} = \text{NEXTHOP}(\zeta_k, \eta)$ не находится в S_0 (т. е. $S_k = S_0$, но $S_{k+1} \neq S_0$). Альтернативно, этот процесс останавливается, если $\zeta_k = \eta$ или ζ_k совпадает с η в своих первых 96 битах.
- [OutboundQueuing] (Очередь исходящих) — Сообщение m включается в $OutMsgDescr$ (с ключом, равным его хэшу), с оболочкой, содержащей его транзитный адрес ζ_k и адрес следующего перехода ζ_{k+1} , как объяснено в 2.1.16 и 2.1.15. То же самое сообщение в оболочке также включается в $OutMsgQueue$ состояния S_k с ключом, равным конкатенации первых 96 бит его адреса следующего перехода ζ_{k+1} (который может быть равен η , если η принадлежит S_k) и хэша сообщения $\text{HASH}(m)$.
- [QueueWait] (Ожидание в очереди) — Сообщение m ждет в $OutMsgQueue$ шардчайна S_k для дальнейшей пересылки. Тем временем, шардчайн S_k может разделиться или объединиться с другими шардчайнами; в этом случае новый шард S'_k , содержащий транзитный адрес ζ_k , наследует m в своей $OutMsgQueue$.
- [ImportInbound] (Импорт входящих) — В какой-то момент в будущем валидаторы для шардчайна S_{k+1} , содержащего адрес следующего перехода ζ_{k+1} , сканируют $OutMsgQueue$ в состоянии шардчайна S_k и решают импортировать сообщение m в соответствии с условием монотонности (см. 2.2.5) и другими условиями. Генерируется новый блок для шардчайна S_{k+1} с включенной копией m в $InMsgDescr$. Запись в $InMsgDescr$ также содержит причину для импорта m в этот блок с хэшем самого последнего блока шардчайна S'_k и предыдущими адресами следующего перехода и транзитными адресами ζ_k и ζ_{k+1} , чтобы соответствующая запись в $OutMsgQueue S'_k$ могла быть легко найдена.
- [Confirmation] (Подтверждение) — Эта запись в $InMsgDescr S_{k+1}$ также служит подтверждением для S'_k . В более позднем блоке S'_k , сообщение m должно быть удалено из $OutMsgQueue S'_k$; это изменение отражается в специальной записи в $OutMsgDescr$ блока S'_k , выполняющего это изменение состояния.
- [Forwarding?] (Пересылка?) — Если конечный адрес назначения η сообщения m не находится в S_{k+1} , сообщение *пересыпается* дальше. Применяется Hypercube

Routing до получения некоторых ζ_l , $l > k$ и $\zeta_{l+1} = \text{NEXTHOP}(\zeta_l, \eta)$, так что ζ_l находится в S_{k+1} , но ζ_{l+1} не находится (см. [2.1.11](#)). После этого копия с новой оболочкой m с установленным транзитным адресом на ζ_l и адресом следующего перехода ζ_{l+1} включается как в $OutMsgDescr$ текущего блока S_{k+1} , так и в $OutMsgQueue$ нового состояния S_{k+1} . Запись об m в $InMsgDescr$ содержит флаг, указывающий, что сообщение было переслано; запись в $OutMsgDescr$ содержит сообщение с новой оболочкой и флаг, указывающий, что это пересланное сообщение. Затем все шаги, начиная с [OutboundQueuing], повторяются, для l вместо k .

- [Processing?] ([Обработка?]) — Если конечный адрес назначения η сообщения m находится в S_{k+1} , то блок S_{k+1} , импортировавший сообщение, должен обработать его с помощью транзакции t , включенной в тот же блок. В этом случае $InMsgDescr$ содержит ссылку на t по его логическому времени $LT(t)$ и флаг, указывающий, что сообщение было обработано.

Приведенный выше алгоритм маршрутизации сообщений не учитывает некоторые дальнейшие модификации, необходимые для реализации Instant Hypercube Routing (IHR). Например, сообщение может быть *отброшено* после импорта (указанного в $InMsgDescr$) в его конечный или промежуточный блок шардчайна, поскольку представлено доказательство доставки через IHR до конечного назначения. В этом случае такое доказательство должно быть включено в $InMsgDescr$ для объяснения, почему сообщение не было дальше переслано или обработано.

2.3 Instant Hypercube Routing и комбинированные гарантии доставки

В этом разделе описывается протокол Instant Hypercube Routing, обычно применяемый TON Blockchain параллельно с ранее упомянутым протоколом Hypercube Routing для достижения более быстрой доставки сообщений. Однако, когда оба протокола Hypercube Routing и Instant Hypercube Routing применяются к одному и тому же сообщению параллельно, обеспечение доставки и гарантий уникальной доставки становится более сложным. Эта тема также обсуждается в этом разделе.

2.3.1. Обзор Instant Hypercube Routing. Позвольте нам объяснить основные шаги, когда механизм Instant Hypercube Routing (IHR) применяется к сообщению. (Обратите внимание, что как обычный HR, так и IHR работают параллельно для одного и того же сообщения; некоторые меры должны быть приняты для обеспечения уникальности доставки любого сообщения.)

Рассмотрим маршрутизацию и доставку того же самого сообщения m с исходным ζ и назначением η , как обсуждалось в [2.2.12](#):

- [NetworkSend] (Отправка по сети) — После того, как валидаторы S_0 согласовали и подписали блок, содержащий транзакцию создания t для m , и убедились, что адрес назначения η для m не находится внутри S_0 , они могут отправить датаграмму (зашифрованное сетевое сообщение), содержащую сообщение m

вместе с доказательством Меркла о его включении в $OutMsgDescr$ только что сгенерированного блока, группе валидаторов шардчейна T , в настоящее время владеющего назначением η .

- [NetworkReceive] (Прием по сети) — Если валидаторы шардчейна T получают такое сообщение, они проверяют его действительность, начиная с самого последнего блока мастерчейна и хэшем блока шардчейна, перечисленных в нем, включая самый последний "канонический" блок шардчейна S_0 . Если сообщение недействительно, они игнорируют его. Если тот блок шардчейна S_0 имеет больший номер последовательности, чем указано в самом последнем блоке мастерчейна, они могут либо отбросить его, либо отложить проверку до появления следующего блока мастерчейна.
- [InclusionConditions] (Условия включения) — Валидаторы проверяют условия включения для сообщения m . В частности, они должны убедиться, что это сообщение не было доставлено ранее и что $OutMsgQueues$ соседей не имеют необработанных исходящих сообщений с адресами назначения в T с меньшим логическим временем создания, чем $LT(m)$.
- [Deliver] (Доставка) — Валидаторы доставляют и обрабатывают сообщение, включая его в $InMsgDescr$ текущего блока шардчейна вместе с битом, указывающим, что это сообщение IHR, доказательством Меркла о его включении в $OutMsgDescr$ оригинального блока и логическим временем транзакции t' , обрабатывающей это входящее сообщение в текущем сгенерированном блоке.
- [Confirm] (Подтверждение) — Наконец, валидаторы отправляют зашифрованные датаграммы всем группам валидаторов промежуточных шардчейнов на пути от ζ к η , содержащие доказательство Меркла о включении сообщения m в $InMsgDescr$ его конечного назначения. Валидаторы промежуточного шардчейна могут использовать это доказательство для удаления копии сообщения m следующего по правилам HR, импортируя сообщение в их $InMsgDescr$ вместе с доказательством Меркла о конечной доставке и установив флаг, указывающий, что сообщение было отброшено.

Общая процедура даже проще, чем для Hypercube Routing. Обратите внимание, однако, что IHR не обеспечивает гарантии доставки или FIFO: сетевое сообщение может быть потеряно при передаче, или валидаторы шардчейна назначения могут решить не действовать на него, или они могут его отбросить из-за переполнения буфера. По этой причине IHR используется как дополнение к HR, а не как замена.

2.3.2. Общие гарантии доставки. Обратите внимание, что комбинация HR и IHR гарантирует конечную доставку любого внутреннего сообщения до его конечного назначения. Действительно, HR сам по себе гарантированно доставит любое сообщение в конце концов, и HR для сообщения m может быть отменен на промежуточном этапе только доказательством Меркла о доставке m до его конечного назначения (через IHR).

2.3.3. Общие гарантии уникальной доставки. Однако *的独特性* доставки сообщений для комбинации HR и IHR最难达到。在某些情况下，可能需要验证以下条件，并且如果必要的话，能够提供简短的 Merkle 证明，表明它们被执行或未执行：

- Когда сообщение m импортируется в свой следующий промежуточный блок шардчайна через HR, мы должны проверить, что m еще не было импортировано через HR.
- Когда m импортируется и обрабатывается в своем конечном шардчайне, мы должны проверить, что m еще не было обработано. Если оно было, существуют три подслучаи:
 - Если m рассматривается для импорта через HR и уже было импортировано через HR, оно не должно быть импортировано снова через HR.
 - Если m рассматривается для импорта через HR и уже было импортировано через IHR (но не через HR), оно должно быть импортировано и немедленно отброшено (без обработки транзакцией). Это необходимо для удаления m из *OutMsgQueue* его предыдущего промежуточного шардчайна.
 - Если m рассматривается для импорта через IHR и уже было импортировано через IHR или HR, оно не должно быть импортировано снова.

2.3.4. Проверка, было ли сообщение уже доставлено до конечного назначения.

Рассмотрим следующий общий алгоритм для проверки, было ли сообщение m уже доставлено до конечного назначения η : можно просто просканировать последние несколько блоков, принадлежащих шардчайну, содержащему адрес назначения, начиная с последнего блока и двигаясь назад через ссылки на предыдущие блоки. (Если есть два предыдущие блока, т. е. если в какой-то момент произошло событие слияния шардчайнов, следуйте цепочке, содержащей адрес назначения.) $InMsgDescr$ каждого из этих блоков можно проверить на наличие записи с ключом $HASH(m)$. Если такая запись найдена, сообщение m уже доставлено, и мы можем легко построить доказательство Меркля этого факта. Если мы не находим такую запись до достижения блока B с $LT^+(B) < LT(m)$, что означает, что m не могло быть доставлено в B или любой из его предшественников, тогда сообщение m определенно еще не было доставлено.

Очевидный недостаток этого алгоритма заключается в том, что, если сообщение m очень старое (и, скорее всего, доставлено давно), что означает, что у него маленькое значение $LT(m)$, потребуется просканировать большое количество блоков, прежде чем получить ответ. Более того, если ответ отрицательный, размер доказательства Меркля этого факта будет линейно увеличиваться с количеством сканируемых блоков.

2.3.5. Проверка, было ли IHR сообщение уже доставлено до конечного назначения.

Чтобы проверить, было ли IHR сообщение m уже доставлено до шардчайна назначения, можно применить общий алгоритм, описанный выше (см. [2.3.4](#)), модифицированный для проверки только последних c блоков для некоторой небольшой константы c (скажем, $c = 8$). Если после проверки этих блоков не удается сделать вывод, то валидаторы для шардчайна назначения могут просто отбросить сообщение IHR вместо того, чтобы тратить больше ресурсов на эту проверку.

2.3.6. Проверка, было ли сообщение HR уже доставлено через HR до конечного назначения или промежуточного шардчайна.

Чтобы проверить, было ли сообщение m , полученное через HR (или, скорее, сообщение m , рассматриваемое для импорта через HR), уже импортировано через HR, можно использовать следующий алгоритм: пусть ζ_k

будет транзитным адресом m (принадлежащим соседнему шардчайну S_k), и ζ_{k+1} является его адресом следующего перехода (принадлежащим рассматриваемому шардчайну). Поскольку мы рассматриваем включение m , то m должно присутствовать в $OutMsgQueue$ самого последнего состояния шардчайна S_k с ζ_k и ζ_{k+1} , указанными в его оболочке. В частности, (a) сообщение было включено в $OutMsgQueue$, и мы можем даже знать, когда, потому что запись в $OutMsgQueue$ иногда содержит логическое время блока, где оно было добавлено, и (b) оно еще не было удалено из $OutMsgQueue$.

Теперь валидаторы соседнего шардчайна обязаны удалить сообщение из $OutMsgQueue$, как только они увидят, что сообщение (с транзитным и адресом следующего перехода ζ_k и ζ_{k+1} в его оболочке) было импортировано в $InMsgDescr$ сообщения шардчайна следующего перехода. Следовательно, (b) подразумевает, что сообщение могло быть импортировано в $InMsgDescr$ предыдущего блока только если этот предыдущий блок очень новый (т. е. еще не известен самому последнему соседнему блоку шардчайна). Поэтому только очень ограниченное количество предыдущих блоков (обычно один или два максимум) нужно сканировать с помощью алгоритма, описанного в [2.3.4](#), чтобы заключить, что сообщение еще не было импортировано.²¹ В действительности, если эту проверку выполняют валидаторы или коллаторы текущего шардчайна, ее можно оптимизировать, сохраняя в памяти $InMsgDescr$ нескольких последних блоков.

2.3.7. Проверка, было ли сообщение HR уже доставлено через IHR до конечного назначения. Наконец, чтобы проверить, было ли сообщение HR уже доставлено до конечного назначения через IHR, можно использовать общий алгоритм, описанный в [2.3.4](#). В отличие от [2.3.5](#) мы не можем прервать процесс проверки после сканирования фиксированного числа последних блоков в шардчайне назначения, потому что сообщения HR не могут быть отброшены без причины.

Вместо этого мы косвенно ограничиваем количество блоков, которые нужно проверить, запрещая включение IHR сообщения m в блок B его шардчайна назначения, если уже есть более чем, скажем, $c = 8$ блоков B' в шардчайне назначения с $LT^+(B') \geq LT(m)$.

Такое условие эффективно ограничивает временной интервал после создания сообщения m , в течение которого оно могло быть доставлено через IHR, так что нужно будет проверить только небольшое количество блоков шардчайна назначения (максимум c).

Обратите внимание, что это условие хорошо согласуется с модифицированным алгоритмом, описанным в [2.3.5](#), эффективно запрещая валидаторам импортировать вновь полученное сообщение IHR, если требуется более чем $c = 8$ шагов для проверки, что оно еще не было импортировано.

²¹ Необходимо не только проверить ключ $HASH(m)$ в $InMsgDescr$ этих блоков, но и проверить промежуточные адреса в оболочке соответствующей записи, если они найдены.

3 Сообщения, дескрипторы сообщений и очереди

Эта глава представляет внутреннюю структуру отдельных сообщений, дескрипторов сообщений (таких как *InMsgDescr* или *OutMsgDescr*) и очередей сообщений (таких как *OutMsgQueue*). Обсуждаются также сообщения с оболочкой (см. [2.1.16](#)).

Обратите внимание, что большинство общих конвенций, связанных с сообщениями, должны соблюдаться всеми шардчейнами, даже если они не принадлежат основному шардчейну; в противном случае, взаимодействие между разными шардчейнами было бы невозможно. Это *интерпретация* содержимого сообщений и обработка сообщений, обычно некоторыми транзакциями, которые различаются между шардчейнами.

3.1 Адрес, валюта и структура сообщений

Эта глава начинается с некоторых общих определений, за которыми следует точная структура адресов, используемых для сериализации исходных и конечных адресов в сообщении.

3.1.1. Некоторые стандартные определения. Для удобства читателя мы представляем здесь несколько общих определений TL-B.²² Эти определения используются для адреса и структуры сообщения, но иначе не связаны с 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. Схема TL-B для адресов. Сериализация исходных и конечных адресов определена следующей схемой TL-B:

²² Описание более старой версии TL может быть найдено на <https://core.telegram.org/mtproto/TL>. В качестве альтернативы неформальное введение в схемы TL-B может быть найдено в [4, 3.3.4].

```

addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 8) external_address:(len * Bit)
    = MsgAddressExt;
anycast_info$_ 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;

```

Последние две строки определяют тип **MsgAddress** как внутренний тип адресов **MsgAddressInt** и **MsgAddressExt** (не путать с их внешним объединением **Either** **MsgAddressInt** **MsgAddressExt**, как определено в [3.1.1](#)), как если бы предыдущие четыре строки были повторены с заменой правой части на **MsgAddress**. Таким образом, тип **MsgAddress** имеет четыре конструктора, и типы **MsgAddressInt** и **MsgAddressExt** являются подтипами **MsgAddress**.

3.1.3. Внешние адреса. Первые два конструктора, **addr_none** и **addr_extern**, используются для исходных адресов "сообщений из ниоткуда" (входящие внешние сообщения) и для адресов назначения "сообщений в никуда" (исходящие внешние сообщения). Конструктор **addr_extern** определяет "внешний адрес", который игнорируется программным обеспечением TON Blockchain (которое рассматривает **addr_extern** как более длинный вариант **addr_none**), но может использоваться внешним программным обеспечением для своих целей. Например, специальный внешний сервис может проверить адрес назначения всех исходящих внешних сообщений, найденных во всех блоках TON Blockchain, и, если в поле **external_addr** присутствует специальное магическое число, разобрать остаток как IP-адрес и UDP-порт или адрес ADNL (TON Network) и отправить датаграмму с копией сообщения на таким образом полученный сетевой адрес.

3.1.4. Внутренние адреса. Оставшиеся два конструктора, **addr_std** и **addr_var**, представляют внутренние адреса. Первый из них, **addr_std**, представляет собой 8-битный идентификатор *workchain_id* (достаточно для мастерчайна и для базового воркчайна) и 256-битный внутренний адрес в выбранном воркчайне. Второй из них, **addr_var**, представляет адреса в воркчайнах с "большим" *workchain_id* или внутренние адреса длиной не равной 256. Оба этих конструктора имеют optionalное значение **anycast**, отсутствующее по умолчанию, которое позволяет выполнять «переписывание адреса» (address rewriting) при наличии.²³

²³ "Address rewriting" — это функция, используемая для реализации "anycast адресов", которая используется в так называемых *больших* или *глобальных* смарт-контрактах (см. [3, 2.3.18]), которые могут иметь экземпляры в нескольких шардчайнах. Когда адрес входящих данных перенаправляется к смарт-контракту, сообщение может быть направлено к смарт-контракту с адресом, совпадающим с адресом назначения на первых *d* битах, где *d* < 32 — это "глубина разделения" смарт-контракта, указанная в поле **anycast.depth** (см. [2.1.7](#)). В противном случае адреса должны совпадать точно.

Валидаторы должны использовать `addr_std` вместо `addr_var` везде, где это возможно, но должны быть готовы принять `addr_var` во входящих сообщениях. Конструктор `addr_var` предназначен для будущих расширений.

Обратите внимание, что `workchain_id` должен быть валидным идентификатором воркчайна, включенным в текущую конфигурацию мастерчайна, и длина внутреннего адреса должна находиться в диапазоне, разрешенном для указанного воркчайна. Например, нельзя использовать `workchain_id = 0` (базовый воркчайн) или `workchain_id = -1` (мастерчайн) с адресами, которые не равны ровно 256 бит.

3.1.5. Представление сумм валюты в Gram. Суммы в Grams выражаются с помощью двух типов, представляющих переменную длину беззнаковых или знаковых целых чисел, плюс тип `Grams`, специально предназначенный для представления не отрицательных сумм нанограммов, следующим образом:

```
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;
```

Если вы хотите представить x нанограмм, выбираете целое число $\ell < 16$ такое, что $x < 2^{8\ell}$, и сериализуете сначала ℓ , как беззнаковое 4-битное целое число, затем само x , как беззнаковое 8-битное целое число. Обратите внимание, что четыре нулевых бита представляют нулевое количество Grams.

Напоминаем (см. [3, A]), что первоначальный общий запас Grams зафиксирован на уровне пяти миллиардов (т. е. $5 * 10^{18} < 2^{63}$ нанограммов), и ожидается, что он будет расти очень медленно. Следовательно, все суммы Grams, с которыми сталкиваются на практике, будут помещаться в беззнаковые или даже знаковые 64-битные целые числа. Валидаторы могут использовать 64-битное целочисленное представление Grams в своих внутренних вычислениях, однако сериализация этих значений блокчейном — это другой вопрос.

3.1.6. Представление коллекций произвольных валют. Напомним, что TON Blockchain позволяет пользователям определять произвольные криптовалюты или токены, помимо Gram, при условии, что выполнены определенные условия. Такие дополнительные криптовалюты определяются 32-битными `currency_ids`. Список определенных дополнительных криптовалют является частью конфигурации блокчайна, хранящейся в мастерчайне.

Когда требуется представить некоторое количество одной или нескольких таких криптовалют, используется словарь (см. [4, 3.3]) с 32-битными `currency_ids` в качестве ключей и значениями `VarUInteger 32`:

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

Значение, прикрепленное к внутреннему сообщению, представлено значением типа **CurrencyCollection**, которое может описывать определенное (не негативное) количество (нано)граммов, а также некоторые дополнительные валюты, если это необходимо. Обратите внимание, что если дополнительные валюты не требуются, **other** сокращается до единственного нулевого бита.

3.1.7. Структура сообщения. Сообщение состоит из его *заголовка*, за которым следует его *тело* или *payload*. Тело по сути произвольно, его интерпретация зависит от целевого смарт-контракта. Заголовок сообщения стандартен и организован следующим образом:

```
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;
```

Смысл этой схемы следующий.

Тип **Message X** описывает сообщение с телом (или payload) типа *X*. Его сериализация начинается с **info** типа **CommonMsgInfo**, которая представлена в трех вариантах: для внутренних сообщений, входящих внешних сообщений и исходящих внешних сообщений соответственно. Все они имеют исходный адрес **src** и адрес назначения **dest**, которые являются внешними или внутренними в зависимости от выбранного конструктора. Помимо этого, внутреннее сообщение может содержать некоторое **value** в Grams и других определенных валютах (см. [3.1.6](#)), и все сообщения, созданные внутри TON Blockchain, имеют логическое время создания **created_lt** (см. [1.4.6](#)) и время создания в формате unixtime **created_at**, которые автоматически устанавливаются создающей транзакцией. Время создания в unixtime равно времени создания блока, содержащего генерирующую транзакцию.

3.1.8. Пересылка и IHR платежи. Общая стоимость внутреннего сообщения.

Внутренние сообщения определяют **ihr_fee** в Grams, которая вычитается из значения, прикрепленного к сообщению, и присуждается валидаторам шардчейна назначения, если они включают сообщение с помощью механизма IHR. **fwd_fee** — это первоначальная общая плата за пересылку, уплаченная за использование механизма HR; она

автоматически рассчитывается на основе некоторых параметров конфигурации и размера сообщения в момент его генерации.

Обратите внимание, что общее значение, переносимая вновь созданным внутренним исходящим сообщением, равна сумме **value**, **ihr_fee** и **fwd_fee**. Эта сумма вычитается из баланса исходного аккаунта. Из этих компонентов только **value** всегда зачисляется на аккаунт назначения при доставке сообщения. **fwd_fee** собирается валидаторами на пути HR от источника к месту назначения, а **ihr_fee** либо собирается валидаторами шардчейна назначения (если сообщение доставляется через IHR), либо зачисляется на аккаунт назначения.

3.1.9. Код и данные, содержащиеся в сообщении. Помимо общей информации о сообщении, хранящейся в **info**, сообщение может содержать части кода и данных смарт-контракта назначения. Эта функция используется, например, в так называемых *сообщениях-конструкторах* (см. [1.7.3](#)), которые являются просто внутренними или входящими внешними сообщениями с определенными полями **code** и, возможно, **data** в **init** частях. Если хэш этих полей верен и у целевого смарт-контракта нет кода или данных, вместо этого используются значения из сообщения.²⁴

3.1.10. Использование code и data в других целях. Воркчайны, кроме мастерчайна и базового воркчайна, могут использовать деревья ячеек, указанные в полях **code**, **data** и **library**, для своих собственных целей. Система сообщений сама по себе не делает предположений о содержимом; они становятся актуальными только когда сообщение обрабатывается транзакцией.

3.1.11. Отсутствие явной цены и лимита на газ. Обратите внимание, что сообщения не имеют явной цены и лимита на газ. Вместо этого цена газа устанавливается глобально валидаторами для каждого воркчайна (это специальный конфигурируемый параметр), и лимит газа для каждой транзакции также имеет стандартное значение, которое является конфигурируемым параметром; смарт-контракт сам может снизить лимит газа во время его выполнения, если это необходимо.

Для внутренних сообщений начальный лимит газа не может превышать значение Gram в сообщении, умноженное на текущую цену газа. Для входящих внешних сообщений начальный лимит газа очень мал, и истинный лимит газа устанавливается самим принимающим смарт-контрактом, когда он *принимает* входящее сообщение с помощью соответствующих примитивов TVM.

²⁴ Более точно, информация из поля **init** входящего сообщения используется либо когда принимающий аккаунт не инициализирован или заморожен с хэшем *StateInit*, ожидаемым аккаунтом, либо когда принимающий аккаунт активен и его код или данные представляют собой внешнюю ссылку на хэш, соответствующий хешу кода или данных, полученных в *StateInit* сообщения.

3.1.12. Десериализация payload сообщения. Payload или тело сообщения десериализуется принимающим смарт-контрактом при выполнении TVM. Система сообщений сама по себе не делает предположений о внутреннем формате payload. Однако имеет смысл описать сериализацию поддерживаемых входящих сообщений по схемам TL или TL-B с 32-битными конструкторскими тегами, чтобы разработчики других смарт-контрактов знали интерфейс, поддерживаемый конкретным смарт-контрактом.

Сообщение всегда сериализуется в блокчейне как последнее поле в ячейке. Поэтому программное обеспечение блокчейна может предположить, что любые биты и ссылки, оставшиеся нераспарсеными после парсинга полей **Message**, предшествующего **body**, принадлежат payload **body** : X , не зная ничего о сериализации типа X .

3.1.13. Сообщения с пустыми payloads. Payload сообщения может быть пустым срезом ячейки, не содержащим битов данных и ссылок. По условию, такие сообщения используются для простых переводов средств. От получающего смарт-контракта обычно ожидается, что он будет обрабатывать такие сообщения тихо и успешно завершать их обработку (с нулевым кодом завершения), хотя некоторые смарт-контракты могут выполнять нетривиальные действия даже при получении сообщения с пустой payload. Например, смарт-контракт может проверить итоговый баланс и, если это достаточно для ранее отложенного действия, инициировать его. Кроме того, смарт-контракт может захотеть сохранить в своем постоянном хранилище полученную сумму и соответствующего отправителя, чтобы позже распределить некоторые токены между каждым отправителем пропорционально переведенным средствам.

Обратите внимание, что даже если смарт-контракт не предпринимает специальных мер для обработки сообщений с пустыми payloads и выбрасывает исключение при их обработке, полученное значение (за вычетом платы за газ) все равно будет добавлена к балансу смарт-контракта.

3.1.14. Исходный адрес сообщения и логическое время создания определяют его генерирующий блок. Обратите внимание, что *исходный адрес и логическое время создания внутреннего или исходящего внешнего сообщения однозначно определяют блок, в котором было сгенерировано сообщение*. Действительно, исходный адрес определяет исходный шардчейн, и блоки этого шардчейна назначаются непересекающимися логическими временными интервалами, так что только один из них может содержать указанное логическое время создания. Вот почему в сообщениях не требуется явное упоминание генерирующего блока.

3.1.15. Сообщения с оболочкой. Оболочки сообщений используются для прикрепления информации о маршрутизации, такой как текущий (транзитный) адрес и адрес следующего перехода, к входящим, транзитным и исходящим сообщениям (см. [2.1.16](#)). Само сообщение хранится в отдельной ячейке и ссылается на оболочку сообщения по ссылке ячейки.

```

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;

```

Тип **IntermediateAddress** используется для описания промежуточных адресов сообщения, то есть для его текущего (или транзитного) адреса **cur_addr** и адреса следующего перехода **next_addr**. Первый конструктор **interm_addr_regular** представляет промежуточный адрес с использованием оптимизации, описанной в [2.1.15](#), сохраняя число первых битов промежуточного адреса, которые такие же, как в исходном адресе; два других явно хранят идентификатор воркчейна и первые 64 бита адреса в этом воркчейне (оставшиеся биты можно взять из исходного адреса). Поле **fwd_fee_remaining** явно представляет максимальное количество средств для пересылки, которое можно вычесть из значения сообщения во время оставшихся шагов HR; оно не может превышать значение **fwd_fee**, указанное в сообщении.

3.2 Дескрипторы входящих сообщений

Этот раздел об *InMsgDescr*, структуре, содержащей описание всех входящих сообщений, импортированных в блок ²⁵

3.2.1. Типы и источники входящих сообщений. Каждое входящее сообщение, упомянутое в *InMsgDescr*, описывается значением типа *InMsg* («дескриптор входящего сообщения»), которое указывает источник сообщения, причину его импорта в этот блок и некоторую информацию о его «судьбе» — его обработке транзакцией или пересылке внутри блока.

Входящие сообщения можно классифицировать следующим образом:

- *Входящие внешние сообщения* — Не требуют дополнительных причин для импорта в блок, но должны быть немедленно обработаны транзакцией в том же блоке.
- *Внутренние IHR сообщения с адресами назначения в этом блоке* — Причина их импорта в блок включает доказательство Меркля их генерации (т. е. их включение в *OutMsgDescr* их исходного блока). Такое сообщение должно быть немедленно доставлено в его конечное место назначения и обработано транзакцией.

²⁵ Строго говоря, *InMsgDescr* является типом этой структуры; мы намеренно используем такую же нотацию для описания этой структуры, чтобы упомянуть только один экземпляр этого типа в блоке.

- *Внутренние сообщения с назначением в этом блоке* — Причина их включения заключается в их присутствии в *OutMsgQueue* последнего состояния соседнего шардчайна²⁶ или в их присутствии в *OutMsgDescr* этого блока. Этот соседний шардчайн полностью определяется транзитным адресом, указанным в оболочке пересылаемого сообщения, который также дублируется в *InMsg*. "Судьба" этого сообщения снова описывается ссылкой на транзакцию обработки в текущем блоке.
- *Немедленно маршрутизуемые внутренние сообщения* — По сути, подкласс предыдущего класса сообщений. В этом случае импортированное сообщение является одним из исходящих сообщений, созданных в этом блоке.
- *Транзитные внутренние сообщения* — Имеют ту же причину для включения, что и предыдущий класс сообщений. Однако они не обрабатываются внутри блока, а внутренне перенаправляются в *OutMsgDescr* и *OutMsgQueue*. Этот факт, наряду со ссылкой на новую оболочку транзитного сообщения, должен быть зарегистрирован в *InMsg*.
- *Отброшенные внутренние сообщения с назначением в этом блоке* — Внутреннее сообщение с назначением в этом блоке может быть импортировано и немедленно отброшено вместо обработки транзакцией, если оно уже было получено и обработано через IHR в предыдущем блоке этого шардчайна. В этом случае должна быть предоставлена ссылка на предыдущую обрабатывающую транзакцию.
- *Отброшенные транзитные внутренние сообщения* — Аналогично, транзитное сообщение может быть немедленно отброшено после импорта, если оно уже было доставлено к своему конечному пункту назначения через IHR. В этом случае требуется доказательство Меркля его обработки в финальном блоке (как сообщение IHR).

3.2.2. Дескриптор входящего сообщения. Каждое входящее сообщение описывается экземпляром типа **InMsg**, который имеет шесть конструкторов, соответствующих случаям, перечисленным выше в [3.2.1](#):

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

    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;
```

²⁶ Напомним, что шардчайн считается соседним сам для себя.

Обратите внимание, что обработка транзакции упоминается в первых четырех конструкторах непосредственно через ссылку на ячейку **Transaction**, хотя логическое время транзакции **transaction_lt:uint64** было бы достаточно для этой цели.

Внутренние условия консенсуса и обеспечивают, что упомянутая транзакция действительно относится к смарт-контракту назначения, указанному в сообщении, и что входящее сообщение, обработанное этой транзакцией, действительно является тем, о котором идет речь в данном экземпляре *InMsg*.

Более того, обратите внимание, что **msg_import_imm** можно отличить от **msg_import_fin**, поскольку это единственный случай, когда логическое время создания обрабатываемого сообщения больше или равно (минимальному) логическому времени блока, импортирующего сообщение.

3.2.3. Сбор платы за пересылку и транзит с импортированных сообщений. Структура *InMsg* также используется для указания платы за пересылку и транзит, собранной с входящих сообщений. Сама плата указывается в полях **ihr_fee**, **fwd_fee** или **transit_fee**; она отсутствует только во входящих внешних сообщениях, которые используют другие механизмы для вознаграждения валидаторов за их импорт. Плата должна удовлетворять следующим внутренним условиям консенсуса:

- Для внешних сообщений (**msg_import_ext**) плата за пересылку не взимается.
- Для внутренних сообщений, импортированных через IHR (**msg_import_ihr**), плата равна **ihr_fee**, которая должна совпадать с указанным в сообщении значением **ihr_fee**. Обратите внимание, что **fwd_fee** или **fwd_fee_remaining** никогда не собираются с сообщений, импортированных через IHR.
- Для внутренних сообщений, доставленных к их назначению (**msg_import_fin** и **msg_import_imm**), плата равна значению **fwd_fee_remaining** во входящем сообщении **in_msg**. Обратите внимание, что она не может превышать значение **fwd_fee**, указанное в самом сообщении.
- Для транзитных сообщений (**msg_import_tr**) плата равна разнице между значениями **fwd_fee_remaining**, указанными во входящем и исходящем оболочках сообщений.
- Для отброшенных сообщений плата также равна **fwd_fee_remaining**, указанной в **in_msg**.

3.2.4. Импортированное значение входящего сообщения. Каждое импортированное сообщение вносит некоторое значение — определенное количество одной или нескольких криптовалют — в блок. Эта импортированное значение рассчитывается следующим образом:

- Внешнее сообщение не импортирует значение.
- Импортированное через IHR сообщение импортирует свое значение плюс его **ihr_fee**.
- Доставленное или транзитное внутреннее сообщение импортирует свое значение, плюс его **ihr_fee**, плюс значение **fwd_fee_remaining** из ее оболочки **in_msg**.
- Отброщенное сообщение импортирует **fwd_fee_remaining** из ее оболочки **in_msg**.

Обратите внимание, что собранные платы за пересылку и транзит с импортированного сообщения не превышают его импортированного значения.

3.2.5. Расширенные хэшмапы или словари.

Перед тем, как продолжить, давайте обсудим сериализацию *расширенных* хэшмапов или словарей.

Расширенные хэшмапы — это структуры хранения ключ-значение с n -битными ключами и значениями некоторого типа X , аналогичные обычным хэшмапам, описанным в [4, 3.3]. Однако каждый промежуточный узел дерева Патриции, представляющий расширенный хэшмап, *расширен* значением типа Y .

Эти значения расширения должны удовлетворять определенным *условиям агрегации*. Обычно Y — это тип целого числа, и условие агрегации заключается в том, что значение расширения разветвления должно равняться сумме значений расширения его двух потомков. В общем случае вместо суммы используется *функция оценки разветвления* $S : Y \times Y \rightarrow Y$ или $S : Y \rightarrow Y \rightarrow Y$. Значение расширения листа обычно вычисляется из значения, хранящегося в этом листе, с помощью *функции оценки листа* $L : X \rightarrow Y$. Значение расширения листа может быть явно сохранено в листе вместе со значением; однако в большинстве случаев в этом нет необходимости, поскольку функция оценки листа L очень проста.

3.2.6. Сериализация расширенных хэшмапов.

Сериализация расширенных хэшмапов с n -битными ключами, значениями типа X и значениями расширения типа Y осуществляется с помощью следующей схемы TL-B, которая является расширением схемы, представленной в [4, 3.3.3]:

```
ahm_edge#_ {n:#} {X:Type} {Y:Type} {l:#} {m:#}
  label:(HmLabel ~l n) {n = (~m) + 1}
  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. Расширение *InMsgDescr*.

Коллекция дескрипторов входящих сообщений расширяется вектором из двух значений валюты, представляющих импортированное значение и сборы за пересылку и транзит, собранные с сообщения или коллекции сообщений:

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

3.2.8. Структура *InMsgDescr*. Теперь *InMsgDescr* сам по себе определяется как расширенный хэшмап с 256-битными ключами (равными хэшам представления импортированных сообщений), значениями типа **InMsg** (см. 3.2.2) и значениями расширения типа **ImportFees** (см. 3.2.7):

_ (HashMapAugE 256 InMsg ImportFees) = InMsgDescr;

Эта нотация TL-B использует анонимный конструктор _, чтобы определить **InMsgDescr** как синоним для другого типа.

3.2.9. Правила агрегации для *InMsgDescr*. Функции оценки разветвления и оценки листа (см. 3.2.5) не включены явно в приведенную выше нотацию, потому что зависимые типы TL-B недостаточно выразительны для этой цели. На словах функция оценки разветвления — это просто покомпонентное сложение двух экземпляров **ImportFees**, а функция оценки листа определяется правилами, перечисленными в 3.2.3 и 3.2.4. Таким образом, корень дерева Патриции, представляющий экземпляр *InMsgDescr*, содержит экземпляр *ImportFees* с общим значением, импортированным всеми входящими сообщениями, и с общими сборами за пересылку, собранными с них.

3.3 Очередь исходящих сообщений и дескрипторы

Этот раздел об *OutMsgDescr*, структуре, представляющей все исходящие сообщения блока, вместе с их оболочками и краткими описаниями причин для включения их в *OutMsgDescr*. Эта структура также описывает все модификации *OutMsgQueue*, которая является частью состояния шардчайна.

3.3.1 Типы исходящих сообщений. Исходящие сообщения можно классифицировать следующим образом:

- *Внешние исходящие сообщения* или «сообщения в никуда» — Генерируются транзакцией внутри этого блока. Причина для включения такого сообщения в *OutMsgDescr* — это просто ссылка на его генерирующую транзакцию.
- *Немедленно обработанные внутренние исходящие сообщения* — Генерируются и обрабатываются в этом самом блоке и не включаются в *OutMsgQueue*. Причина для включения такого сообщения — это ссылка на его генерирующую транзакцию, а его «судьба» описывается ссылкой на соответствующую запись в *InMsgDescr*.
- *Обычные (внутренние) исходящие сообщения* — Генерируются в этом блоке и включаются в *OutMsgQueue*.
- *Транзитные (внутренние) исходящие сообщения* — Импортируются в *InMsgDescr* того же блока и маршрутизируются через HR до тех пор, пока не будет получен адрес следующего перехода за пределами текущего шардчайна.

3.3.2. Записи об извлечении сообщений из очереди. Помимо вышеуказанных типов исходящих сообщений *OutMsgDescr* может содержать специальные «записи об извлечении сообщений из очереди», которые указывают, что сообщение было удалено из *OutMsgQueue* в этом блоке. Причина этого удаления указывается в записи удаления

сообщения; она состоит из ссылки на сообщение в оболочке, которое удаляется, и логического времени блока соседнего шардчайна, который имеет это сообщение в оболочке в своем *InMsgDescr*.

Обратите внимание, что в некоторых случаях сообщение может быть импортировано из *OutMsgQueue* текущего шардчайна, направлено внутренне, а затем снова включено в *OutMsgDescr* и *OutMsgQueue* с другой оболочкой.²⁷ В этом случае используется вариант описания транзитного исходящего сообщения, который также выполняет функцию записи об удалении сообщения из очереди.

3.3.3. Дескриптор исходящего сообщения. Каждое исходящее сообщение описывается экземпляром *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;
```

Последние два описания имеют эффект удаления (извлечения) сообщения из *OutMsgQueue* вместо его вставки. Последнее повторно вставляет сообщение в *OutMsgQueue* с новой оболочкой после выполнения внутренней маршрутизации (см. [2.1.11](#)).

3.3.4. Экспортируемое значение исходящего сообщения. Каждое исходящее сообщение, описанное *OutMsg*, экспортирует некоторое значение — определенное количество одной или нескольких криптовалют — из блока. Эта экспортируемая стоимость рассчитывается следующим образом:

- Внешнее исходящее сообщение не экспортирует значение.
- Внутреннее сообщение, сгенерированное в этом блоке, экспортирует свое **value**, плюс свою **ihr_fee**, плюс свою **fwd_fee**. Обратите внимание, что **fwd_fee** должна быть равна значению **fwd_fee_remaining**, указанному в оболочке **out_msg**.

²⁷ Эта ситуация редка и происходит только после объединения шардчайнов. Обычно сообщения, импортированные из *OutMsgQueue* того же шардчайна, имеют назначения внутри этого шардчайна и обрабатываются соответствующим образом вместо повторного помещения в очередь.

- Транзитное сообщение экспортирует свое **value**, плюс свою **ihr_fee**, плюс значение **fwd_fee_remaining** из своей оболочки **out_msg**.
- То же самое относится к **msg_export_tr_req**, конструктору *OutMsg*, используемому для повторно вставленных извлеченных сообщений.
- Запись об извлечении сообщения из очереди (**msg_export_deq**; см. [3.3.2](#)) не экспортирует значение.

3.3.5. Структура *OutMsgDescr*. *OutMsgDescr* сам по себе просто расширенный хэшмап (см. [3.2.5](#)) с 256-битными ключами (равными хэшу представления сообщения), значениями типа *OutMsg* и значениями расширения типа *CurrencyCollection*:

_ (HashmapAugE 256 OutMsg CurrencyCollection) = OutMsgDescr;

Расширение — это экспортруемое значение соответствующего сообщения, агрегированное с помощью суммы и вычисленное в листьях, как объяснено в [3.3.4](#). Таким образом, общее экспортруемое значение появляется около корня дерева Патриции, представляющего *OutMsgDescr*.

Наиболее важным условием консенсуса для *OutMsgDescr* является то, что его запись с ключом k должна быть *OutMsg*, описывающим сообщение m с хэшем представления $\text{HASH}^b(m) = k$.

3.3.6. Структура *OutMsgQueue*. Напомним (см. [1.2.7](#)), что *OutMsgQueue* является частью состояния блокчейна, а не блока. Следовательно блок содержит только хеш-ссылки на его начальное и конечное состояние и вновь созданные ячейки.

Структура *OutMsgQueue* проста: это просто расширенный хэшмап с 352-битными ключами и значениями типа *OutMsg*:

_ (HashmapAugE 352 OutMsg uint64) = OutMsgQueue;

Ключ, используемый для исходящего сообщения m — это конкатенация его 32-битного идентификатора следующего перехода *workchain_id*, первых 64 бит адреса следующего перехода внутри этого воркчейна и хэша представления $\text{HASH}^b(m)$ самого сообщения m . Расширение осуществляется логическим временем создания $\text{LT}(m)$ сообщения m на листьях и минимумом значений расширения дочерних значений на развилках.

Наиболее важное условие консенсуса для *OutMsgQueue* заключается в том, что значение по ключу k должно действительно содержать сообщение с оболочкой с ожидаемым адресом следующего перехода и хэшем представления.

3.3.7. Условия консенсуса для *OutMsg*. Несколько внутренних условий консенсуса налагаются на экземпляры *OutMsg*, присутствующие в *OutMsgDescr*. Они включают следующее:

- Каждый из первых трех конструкторов описания исходящего сообщения включает ссылку на генерирующую транзакцию. Эта транзакция должна

принадлежать исходному аккаунту сообщения, она должна содержать ссылку на указанное сообщение как на одно из своих исходящих сообщений и она должна быть восстанавливаемой по ее **account_id** и **transaction_id**.

- **msg_export_tr** и **msg_export_tr_req** должны ссылаться на экземпляр *InMsg*, описывающий то же сообщение (в другой исходной оболочке).
- Если используется один из первых четырех конструкторов, сообщение должно отсутствовать в начальной *OutMsgQueue* блока; в противном случае оно должно присутствовать.
- Если используется **msg_export_deq**, сообщение должно отсутствовать в конечной *OutMsgQueue* блока; в противном случае оно должно присутствовать.
- Если сообщение не упоминается в *OutMsgDescr*, оно должно быть таким же в начальной и конечной *OutMsgQueue* блока.

4 Аккаунты и транзакции

Эта глава о структуре *аккаунтов* (или смарт-контрактов) и их *состоянии* в ТОН Blockchain. Также рассматриваются *транзакции*, которые являются единственным способом изменения состояния аккаунта и обработки входящих сообщений и генерации исходящих сообщений.

4.1 Аккаунты и их состояния

Напомним, что *смарт-контракт* и *аккаунт* — это одно и то же в контексте ТОН Blockchain — и что эти термины могут использоваться взаимозаменяющими, по крайней мере пока рассматриваются только малые (или «обычные») смарт-контракты. Большой смарт-контракт может использовать несколько аккаунтов, расположенных в разных шардчейнах одного и того же воркчайна для балансировки нагрузки.

Аккаунт *идентифицируется* его полным адресом и *полностью описывается* своим состоянием. Другими словами, в аккаунте нет ничего, кроме его адреса и состояния.

4.1.1. Адреса аккаунта. В общем, аккаунт полностью определяется его *полным адресом*, состоящим из 32-битного *workchain_id* и (обычно 256-битного) *внутреннего адреса* или *идентификатора аккаунта account_id* внутри выбранного воркчайна. В базовом воркчайне (*workchain_id = 0*) и в мастерчайне (*workchain_id = -1*) внутренний адрес всегда 256-битный. В этих воркчейнах²⁸ *account_id* не может выбираться произвольно, но должен соответствовать хэшу начального кода и данных смарт-контракта; в противном случае, будет невозможно инициализировать аккаунт с предполагаемым кодом и данными (см. [1.7.3](#)) и сделать что-либо с накопленными средствами на балансе аккаунта.

4.1.2. Нулевой аккаунт. По традиции, *нулевой аккаунт* или *аккаунт с нулевым адресом* накапливает обработку, пересылку и транзитные сборы, а также любые другие платежи, собранные валидаторами мастерчайна или воркчайна. Более того, нулевой аккаунт является "большим смарт-контрактом", что означает, что каждый шардчайн имеет свою инстанцию нулевого аккаунта с наиболее значимыми битами адреса, скорректированными так, чтобы они соответствовали шарду. Любые средства, переведенные на нулевой аккаунт намеренно или случайно фактически являются подарком для валидаторов. Например, смарт-контракт может уничтожить себя, отправив все свои средства на нулевой аккаунт.

²⁸ Для простоты, мы иногда рассматриваем мастерчайн как еще один воркчайн с *workchain_id = -1*.

4.1.3. Маленькие и большие смарт-контракты. По умолчанию смарт-контракты являются «маленькими», что означает, что у них есть один адрес аккаунта, принадлежащий точно одному шардчейну в любой момент времени. Однако можно создать «большой смарт-контракт с глубиной разделения d », что означает, что может быть создано до 2^d экземпляров смарт-контракта, при этом первые d бит исходного адреса смарт-контракта заменяются произвольными битовыми последовательностями.²⁹ Можно отправлять сообщения таким смарт-контрактам, используя внутренние anycast-адреса с **anycast** в d (см. [3.1.2](#)). Кроме того, экземпляры большого смарт-контракта могут использовать этот anycast-адрес в качестве исходного адреса для создаваемых сообщений.

Экземпляр большого смарт-контракта — это аккаунт с ненулевой *максимальной глубиной разделения* d .

4.1.4. Три вида аккаунтов. Существуют три вида аккаунтов:

- *Неинициализированный* — На аккаунте есть только баланс; его код и данные еще не были инициализированы.
- *Активный* — Код и данные аккаунта также были инициализированы.
- *Замороженный* — Код и данные аккаунта заменены хэшем, но баланс все еще хранится явно. Баланс замороженного аккаунта может эффективно стать отрицательным из-за платежей за хранение.

4.1.5. Профиль хранения аккаунта. *Профиль хранения* аккаунта — это структура данных, описывающая количество постоянного хранилища состояния блокчейна, используемого этим аккаунтом. Он описывает общее количество используемых ячеек, бит данных и внутренних и внешних ссылок.

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

Тот же тип **StorageUsed** может представлять профиль хранения сообщения, когда это необходимо, например, для расчета **fwd_fee**, общей комиссии за пересылку для Hypercube Routing. Профиль хранения аккаунта имеет некоторые дополнительные поля, указывающие, когда последний раз были вычтены комиссии за хранение:

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

²⁹ Фактически, до первых d бит заменены таким образом, что каждый шард содержит не более одного экземпляра большого смарт-контракта, и шарды (w, s) с префиксом s длиной $|s| \leq d$ содержат только один экземпляр.

Поле **last_paid** содержит либо время последнего сбора комиссии за хранение в формате unixtime (обычно это время последней транзакции), либо время создания аккаунта в формате unixtime (снова по транзакции). Поле **due_payment**, если оно присутствует, накапливает платежи за хранение, которые не могли быть взысканы с баланса аккаунта, представленного строго положительным количеством нанограммов; оно может присутствовать только для неинициализированных или замороженных аккаунтов, которые имеют баланс ноль в Grams (но могут иметь ненулевые балансы в других криптовалютах). Когда **due_payment** превышает значение конфигурируемого параметра блокчейна, аккаунт полностью уничтожается и его баланс, если таковой имеется, переводится на нулевой аккаунт.

4.1.6. Описание аккаунта. Состояние аккаунта представлено экземпляром типа *Account*, описанного следующей схемой TL-B: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;

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;
```

³⁰ Эта схема использует анонимные конструкторы и анонимные поля, оба обозначены символом подчеркивания _.

Обратите внимание, что **account_frozen** содержит хэш состояния экземпляра *StateInit*, вместо самого экземпляра, который в противном случае содержался бы в **account_active**; **account_uninit** похож на **account_frozen**, но не содержит явного **state_hash**, так как предполагается, что он равен внутреннему адресу аккаунта (*account_id*), который уже присутствует в поле **addr**. Поле **split_depth** присутствует и ненулевое только в экземплярах больших смарт-контрактов. Поле **special** может присутствовать только в мастерчайне и в пределах мастерчайна, в некоторых фундаментальных смарт-контрактах, необходимых для функционирования всей системы.

Статистика хранения, хранящаяся в **storage_stat**, отражает общее использование хранилища среза ячейки **storage**. В частности, биты и ячейки, используемые для хранения баланса, также отражены в **storage_stat**.

Когда необходимо представить несуществующий аккаунт, используется конструктор **account_none**.

4.1.7. Состояние аккаунта как сообщение от аккаунта к своему будущему "я".

Обратите внимание, что состояние аккаунта очень похоже на сообщение, отправленное аккаунтом его будущему "я", участвующему в следующей транзакции, по следующим причинам:

- Состояние аккаунта не меняется между двумя последовательными транзакциями того же аккаунта, поэтому оно полностью схоже в этом отношении с сообщением, отправленным от предыдущей транзакции к последующей.
- Когда транзакция обрабатывается, ее входы — это входящее сообщение и предыдущее состояние аккаунта; ее выходы — это созданные исходящие сообщения и следующее состояние аккаунта. Если рассматривать состояние как особый вид сообщения, мы видим, что каждая транзакция имеет ровно два входа (состояние аккаунта и входящее сообщение) и по крайней мере один выход.
- И сообщение, и состояние аккаунта могут содержать код и данные в экземпляре *StateInit*, а также некоторое значение в их **balance**.
- Аккаунт инициализируется *сообщением-конструктором*, которое по сути несет будущее состояние и баланс аккаунта.
- Иногда сообщения преобразуются в состояния аккаунтов и наоборот. Например, когда происходит событие слияния шардчайнов и два аккаунта, являющиеся экземплярами одного большого контракта, должны быть объединены, один из них преобразуется в сообщение, отправляемое другому (см. [4.2.11](#)). Аналогично, когда происходит событие разделения шардчайна и необходимо разделить большой смарт-контракт на два, это достигается специальной транзакцией, которая создает новый экземпляр с помощью *сообщения-конструктора*, отправленного из существующего экземпляра в новый (см. [4.2.10](#)).
- Можно сказать, что сообщение участвует в передаче некоторой информации *сквозь пространство* (между различными шардчайнами или по крайней мере между аккаунтчайнами), в то время как состояние аккаунта передает информацию *сквозь время* (от прошлого к будущему того же аккаунта).

4.1.8. Различия между сообщениями и состояниями аккаунтов. Конечно, есть и важные различия. Например:

- Состояние аккаунта передается только "во времени" (для блока шардчайна к его преемнику), но никогда "в пространстве" (от одного шардчайна к другому). В результате этот перенос осуществляется неявно, без создания полных копий состояния аккаунта где-либо в блокчейне.
- Плата за хранение, собранная валидаторами за поддержание состояния аккаунта, обычно значительно меньше, чем комиссии за пересылку сообщений для того же объема данных.
- Когда входящее сообщение доставляется аккаунту, вызывается код с этого аккаунта, а не код из сообщения.

4.1.9. Объединенное состояние всех аккаунтов в шарде. Разделенная часть общего состояния шардчайна (см. [1.2.1](#) и [1.2.2](#)) задается следующим образом:

_ (HashMapAugE 256 Account CurrencyCollection) = ShardAccounts;

Это просто словарь с 256-битными *account_ids* в качестве ключей и соответствующими состояниями аккаунтов в качестве значений, расширенных суммой балансов аккаунтов. Таким образом, сумма балансов всех аккаунтов в шардчайне вычисляется, чтобы можно было легко проверить общее количество криптовалюты, «хранящейся» в шарде.

Внутренние условия консенсуса обеспечивают, что адрес аккаунта, на который ссылается ключ *k* в *SmartAccounts*, действительно равен *k*. Дополнительное внутреннее условие консенсуса требует, чтобы все ключи *k* начинались с префикса шарда *s*.

4.1.10. Владелец аккаунта и описания интерфейсов. Можно включить некоторую дополнительную информацию в управляемый аккаунт. Например, индивидуальный пользователь или компания могут добавить текстовое описание к своему кошельковому аккаунту с именем или адресом пользователя или компании (или их хэшем, если информация не должна быть публично доступной). В качестве альтернативы смарт-контракт может предложить машинно-читаемое или читаемое человеком описание своих поддерживаемых методов и их предполагаемого применения, которое может быть использовано продвинутыми приложениями кошелька для создания выпадающих меню и форм, помогающих человеку создать валидные сообщения для отправки этому смарт-контракту.

Один из способов включения такой информации — зарезервировать, скажем, вторую ссылку в ячейке данных состояния аккаунта для словаря с 64-битными ключами (соответствующими некоторым идентификаторам стандартных типов дополнительных данных, которые могут храниться) и соответствующими значениями. Затем блокчейн-explorer сможет извлечь требуемое значение вместе с доказательством Меркла, если это необходимо.

Лучший способ сделать это — определить некоторые *get-методы* в смарт-контракте.

4.1.11. Get-методы смарт-контракта. *Get-методы* выполняются как отдельный экземпляр TVM с загруженным кодом и данными аккаунта. Необходимые параметры передаются через стек (например, магическое число, указывающее поле для извлечения, или конкретный get-метод, который должен быть вызван), и результаты возвращаются на стеке TVM также (например, ячейка с сериализацией строки с именем владельца аккаунта).

Как бонус, get-методы могут использоваться для получения ответов на более сложные запросы, чем просто извлечение постоянного объекта. Например, смарт-контракты TON DNS предоставляют get-методы для поиска строки домена в реестре и возвращения соответствующей записи, если она найдена.

По соглашению get-методы используют большие *отрицательные* 32-битные или 64-битные индексы или магические числа, а внутренние функции смарт-контракта используют последовательные *положительные* индексы, которые применяются в инструкции **CALLDICT** TVM. Функция **main()** смарт-контракта, используемая для обработки входящих сообщений в обычных транзакциях, всегда имеет индекс ноль.

4.2 Транзакции

Согласно Infinite Sharding Paradigm и модели акторов три основных компонента TON Blockchain — это *аккаунты* (вместе с их состояниями), *сообщения* и *транзакции*. В предыдущих разделах уже обсудили первые два; этот раздел рассматривает транзакции.

В отличие от сообщений, которые имеют по существу те же заголовки на протяжении всего TON Blockchain, и аккаунтов, у которых по крайней мере есть общие части (адрес и баланс), наше обсуждение транзакций неизбежно ограничено мастерчайном и базовым шардчайном. Другие шардчайны могут определять совершенно другие виды транзакций.

4.2.1 Логическое время транзакции. Каждая транзакция t имеет логический временной интервал $LT^*(t) = [LT^-(t), LT^+(t)]$, присвоенный ей (см. [1.4.6](#) и [1.4.3](#)). По соглашению транзакция t , генерирующая n исходящих сообщений m_1, \dots, m_n , получает логический временной интервал длиной $n + 1$, так что:

$$LT^+(t) = LT^-(t) + n + 1 \quad . \quad (16)$$

Мы также определяем $LT(t) := LT^-(t)$ и присваиваем логическое время создания сообщения m_i , где $1 \leq i \leq n$, следующим образом:

$$LT(m_i) = LT^-(m_i) := LT^-(t) + i, \quad LT^+(m_i) := LT^-(m_i) + 1 \quad . \quad (17)$$

Таким образом, каждому сгенерированному исходящему сообщению присваивается свой собственный интервал времени в рамках логического временного интервала $LT^*(t)$ транзакции t .

4.2.2 Логическое время уникально идентифицирует транзакции и исходящие сообщения аккаунта. Напомним, что условия, наложенные на логическое время, предполагают, что $LT^-(t) \geq LT^+(t')$ для любой предыдущей транзакции t' того же аккаунта ζ и что $LT^-(t) > LT(m)$, если m — это входящее сообщение, обрабатываемое транзакцией t . Таким образом, логические временные интервалы транзакций одного и того же аккаунта не пересекаются, и, как следствие, логические временные интервалы всех исходящих сообщений, сгенерированных аккаунтом, также не пересекаются. Другими словами, все $LT(m)$ различны, когда m проходит через все исходящие сообщения одного и того же аккаунта ζ .

Таким образом, $LT(t)$ и $LT(m)$, когда они объединены с идентификатором аккаунта ζ , однозначно определяют транзакцию t или исходящее сообщение m того аккаунта. Более того, если у кого-то есть упорядоченный список всех транзакций аккаунта вместе с их логическими временами, легко найти транзакцию, которая сгенерировала данное исходящее сообщение m , просто найдя транзакцию t с логическим временем $LT(t)$, ближайшим к $LT(m)$ снизу.

4.2.3. Общие компоненты транзакции. Каждая транзакция t содержит следующие данные или косвенно к ним относится:

- Аккаунт ζ , которому принадлежит транзакция.
- Логическое время $LT(t)$ транзакции.
- Одно или ноль входящих сообщений m , обработанных транзакцией.
- Количество сгенерированных исходящих сообщений $n \geq 0$.
- Исходящие сообщения m_1, \dots, m_n .
- Начальное состояние аккаунта ζ (включая его баланс).
- Конечное состояние аккаунта ζ (включая его баланс).
- Общая сумма сборов валидаторов.
- Подробное описание транзакции, содержащее всю или некоторую информацию, необходимую для ее проверки, включая вид транзакции (см. 4.2.4) и некоторые промежуточные шаги, которые были выполнены.

Из этих компонентов все, кроме самого последнего, довольно общие и могут встречаться и в других воркчейнах.

4.2.4. Виды транзакций. Существуют различные виды транзакций, разрешенные в мастерчайне и шардчайнах. *Обычные* транзакции заключаются в доставке одного входящего сообщения на аккаунт и его обработке кодом этого аккаунта; это самый распространенный вид транзакций. Кроме того, существует несколько видов *экзотических* транзакций.

Всего существует шесть видов транзакций:

- *Обычные транзакции* — Принадлежат аккаунту ζ . Они точно обрабатывают одно входящее сообщение m (описанное в $InMsgDescr$ включающего блока) с адресом назначения ζ , вычисляют новое состояние аккаунта и генерируют несколько исходящих сообщений (зарегистрированных в $OutMsgDescr$) с источником ζ .
- *Транзакции хранения* — Могут быть вставлены валидаторами по их усмотрению. Они не обрабатывают входящие сообщения и не вызывают код. Их единственное воздействие — сбор платежей за хранение с аккаунта, влияющих на статистику хранения и баланс. Если баланс аккаунта упадет ниже определенной суммы, аккаунт может быть заморожен, а его код и данные заменены их объединенным хэшем.
- *Tick-транзакции* — Автоматически вызываются для определенных специальных аккаунтов (смарт-контрактов) в мастерчайне, которые имеют установленный флаг **tick**. Это самые первые транзакции в каждом блоке мастерчайна. У них нет входящих сообщений, но они могут генерировать исходящие сообщения и изменять состояние аккаунта. Например, *выборы валидаторов* проводятся tick-транзакциями специальных смарт-контрактов в мастерчайне.
- *Tock-транзакции* — Аналогично автоматически вызываются как самые последние транзакции каждого блока мастерчайна для определенных специальных аккаунтов.
- *Разделяющие транзакции* — Вызываются как последние транзакции блоков шардчайна, непосредственно предшествующих событию разделения шардчайна. Они автоматически запускаются для экземпляров больших смарт-контрактов, которые нуждаются в создании нового экземпляра после разделения.
- *Объединяющие транзакции* — Аналогично вызываются как первые транзакции блоков шардчайна сразу после события слияния шардчайна, если экземпляр большого смарт-контракта нуждается в объединении с другим экземпляром того же смарт-контракта.

Обратите внимание, что из этих шести видов транзакций только четыре могут происходить в мастерчайне, и другой набор из четырех может происходить в базовом воркчайне.

4.2.5. Фазы обычной транзакции. Обычная транзакция выполняется в несколько *фаз*, которые можно рассматривать как несколько "под транзакций", тесно связанных в одну:

- *Фаза хранения* — Собирает необходимые платежи за хранение состояния аккаунта (включая код смарт-контракта и данные, если они присутствуют) на данный момент времени. Аккаунт в результате может быть *заморожен*. Если смарт-контракт ранее не существовал, фаза хранения отсутствует.
- *Фаза кредитования* — Аккаунт кредитуется на сумму полученного входящего сообщения.
- *Вычислительная фаза* — Код смарт-контракта вызывается внутри экземпляра TVM с соответствующими параметрами, включая копию входящего сообщения и постоянные данные, и завершается с выходным кодом, новыми постоянными данными и *списком действий* (в который входят, например, исходящие сообщения для отправки). Эта фаза обработки может привести к созданию нового аккаунта (неинициализированного или активного) или к активации ранее

- неинициализированного или замороженного аккаунта. *Расход газа*, равный произведению цены на газ и потребленного газа, вычитается из баланса аккаунта.
- **Фаза действия** — Если смарт-контракт успешно завершен (с выходным кодом 0 или 1), выполняются действия из списка. Если выполнить все действия невозможно, например, из-за недостатка средств для перевода с исходящим сообщением, транзакция прерывается и состояние аккаунта возвращается к исходному. Транзакция также прерывается, если смарт-контракт не завершился успешно или если его вообще невозможно вызвать, потому что он не инициализирован или заморожен.
 - **Фаза отскока** — Если транзакция была прервана, и входящее сообщение имеет установленный **bounce** флаг, оно "отскакивает" путем автоматической генерации исходящего сообщения (со снятым bounce флагом) его оригинальному отправителю. Почти вся стоимость оригинального входящего сообщения (за исключением платы за газ и пересылку) переводится на сгенерированное сообщение, которое в остальном имеет пустое тело.

4.2.6. Отскок входящих сообщений на несуществующие аккаунты. Обратите внимание, что если входящее сообщение с установленным bounce флагом отправляется на ранее не существующий аккаунт и транзакция прерывается (например, потому что в сообщении нет кода и данных с правильным хэшем, так что виртуальная машина вообще не вызывается), то аккаунт не создается, даже как неинициализированный, так как он будет иметь нулевой баланс и не будет содержать кода и данных в любом случае.³¹

4.2.7. Обработка входящего сообщения разделена между вычислительной фазой и фазой действия. Обратите внимание, что обработка входящего сообщения фактически разделена на две фазы: *вычислительную фазу* и *фазу действия*. Во время вычислительной фазы вызывается виртуальная машина и выполняются необходимые вычисления, но никаких действий вне виртуальной машины не предпринимается. Другими словами, *выполнение смарт-контракта в TVM не имеет побочных эффектов*; смарт-контракт не может взаимодействовать с блокчейном напрямую во время своего выполнения. Вместо этого примитивы TVM, такие как SENDMSG, просто сохраняют требуемое действие (например, исходящее сообщение для отправки) в список действий, постепенно накапливающийся в регистре управления TVM с 5. Сами действия откладываются до фазы действия, в течение которой пользовательский смарт-контракт вообще не вызывается.

4.2.8. Причины разделения обработки на вычислительную фазу и фазу действия. Некоторые причины такого разделения:

³¹ В частности, если пользователь ошибочно отправляет средства на несуществующий адрес в *bounceable* сообщении, средства не будут потрачены, а будут возвращены (*bounced*) назад. Поэтому, приложение пользователя должно устанавливать **bounce** флаг во всех генерируемых сообщениях по умолчанию, если не указано иное. Однако сообщения без возможности отскока необходимы в некоторых ситуациях (см. [1.7.6](#)).

- Проще прервать транзакцию, если она завершается с кодом выхода, отличным от 0 или 1.
- Правила обработки выходных действий могут быть изменены без модификации виртуальной машины. (Например, могут быть введены новые выходные действия).
- Сама виртуальная машина может быть изменена или даже заменена другой (например, в новом воркчейне) без изменения правил обработки выходных действий.
- Исполнение смарт-контракта внутри виртуальной машины полностью изолировано от блокчайна и является *чистым вычислением*. В результате это исполнение может быть *виртуализировано* внутри виртуальной машины с помощью примитива TVM **RUNVM**, что полезно для смарт-контрактов валидаторов и смарт-контрактов, управляющих платежными каналами и другими сайдчейнами. Дополнительно виртуальная машина может быть *эмулирована* внутри себя или в упрощенной версии себя, что является полезной функцией для проверки исполнения смарт-контрактов внутри TVM.³²

4.2.9. Транзакции хранения, *tick* и *tock* транзакции. Транзакции хранения очень похожи на отдельную фазу хранения обычной транзакции. Tick и tock транзакции похожи на обычные транзакции без фаз кредита и отскока, поскольку в них нет входящих сообщений.

4.2.10. Разделяющие транзакции. Разделяющие транзакции на самом деле состоят из двух транзакций. Если аккаунт ζ должен быть разделен на два аккаунта ζ и ζ' :

- Сначала выполняется *транзакция подготовки разделения*, похожая на tock-транзакцию (но в шардчейне, а не в мастерчейне), для аккаунта ζ . Это должна быть последняя транзакция для ζ в блоке шардчайна. Результаты этапа обработки транзакции подготовки разделения включают не только новое состояние аккаунта ζ , но и новое состояние аккаунта ζ' , представленное сообщением-конструктором к ζ' (см. [4.1.7](#)).
- Затем добавляется *транзакция установки разделения* для аккаунта ζ' со ссылкой на соответствующую транзакцию подготовки разделения. Транзакция установки разделения должна быть единственной транзакцией для ранее не существующего аккаунта ζ' в блоке. Эффективно она устанавливает состояние ζ' так, как это определено транзакцией подготовки разделения.

4.2.11. Объединяющие транзакции. Объединяющие транзакции также состоят из двух действий. Если аккаунт ζ' нуждается в объединении с аккаунтом ζ :

³² Ссылка на реализацию эмулятора TVM, работающего в упрощенной версии TVM, может быть включена в мастерчейн для использования, когда возникает разногласие между валидаторами по поводу определенного запуска TVM. Таким образом, недостатки в реализации TVM могут быть обнаружены. Ссылка на реализацию служит как авторитетный источник по операционной семантике TVM (см. [4, B.2]).

- Сначала выполняется транзакция подготовки объединения для ζ , которая преобразует все его постоянное состояние и баланс в специальное сообщение-конструктор с адресом назначения ζ (см. [4.1.7](#)).
- Затем транзакция установки объединения для ζ , ссылающаяся на соответствующую транзакцию подготовки объединения, обрабатывает это сообщение-конструктор. Транзакция установки объединения аналогична tick-транзакции, в том смысле, что она должна быть первой транзакцией для ζ в блоке, но она расположена в шардчайне, а не в мастерчайне, и имеет специальное входящее сообщение.

4.2.12. Сериализация общей транзакции. Любая транзакция содержит поля, перечисленные в [4.2.3](#). Следовательно, есть некоторые общие компоненты во всех транзакциях:

```
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;
```

Восклицательный знак в декларации TL-B для **merkle_update** указывает на специальную обработку, необходимую для таких значений. В частности, они должны храниться в отдельной ячейке, которая должна быть помечена как **экзотическая**, как указано в ее заголовке (см. [\[4, 3.1\]](#)).

Полное объяснение сериализации *TransactionDescr*, которая описывает одну транзакцию в соответствии с ее видом, указанным в [4.2.4](#), можно найти в разделе [4.3](#).

4.2.13. Представление исходящих сообщений, генерируемых транзакцией.

Исходящие сообщения, генерируемые транзакцией t , хранятся в словаре **out_msgs** с 15-битными ключами, равными $0, 1, \dots, n - 1$, где $n = \text{outmsg_cnt}$ — это количество сгенерированных исходящих сообщений. Сообщение m_{i+1} с индексом $0 \leq i < n$ должно иметь $\text{LT}(m_{i+1}) = \text{LT}(t) + i + 1$, и $\text{LT}(t) = \text{LT}^-(t)$ явно хранится в поле **lt**.

4.2.14. Условия консенсуса для транзакций. Общая сериализация полей, присутствующая в *Transaction*, независимо от ее типа и описания, позволяет нам накладывать несколько «внешних» условий консенсуса на любую транзакцию. Самое важное из них касается *потока значений* внутри транзакции: сумма всех входов (значение импорта входящего сообщения + исходный баланс аккаунта) должна равняться сумме всех выходов (итоговый баланс аккаунта + сумма всех выходных значений всех отправляемых сообщений + комиссии, взимаемые валидаторами). Таким образом, поверхностный осмотр транзакции, которая обрабатывает входящее сообщение со значением импорта 1 Gram, полученный аккаунтом с начальным балансом в 10 Grams,

исходящее сообщение со значением экспорта 100 Grams в процессе, раскроет его недействительность даже до проверки всех деталей выполнения TVM.

Другие условия консенсуса могут немного зависеть от описания транзакции. Например, входящее сообщение, обработанное обычной транзакцией, должно быть зарегистрировано в *InMsgDescr* охватывающего блока, и соответствующая запись должна содержать ссылку на эту транзакцию. Аналогично, все исходящие сообщения, сгенерированные всеми транзакциями (за исключением одного специального сообщения, сгенерированного транзакцией подготовки разделения или объединения), должны быть зарегистрированы в *OutMsgDescr*.

4.2.15. Коллекция всех транзакций аккаунта. Все транзакции одного аккаунта ξ собираются в "блок аккаунтчайна" *AccountBlock*, который по сути является словарем **transactions** с 64-битными ключами, каждый из которых соответствует логическому времени соответствующей транзакции:

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

Словарь **transactions** расширен значением в *Grams*, которое агрегирует общие сборы с этих транзакций.

В дополнение к этому словарю *AccountBlock* содержит обновление Меркля (см. [4, 3.1]) общего состояния аккаунта. Если аккаунт не существовал до этого блока, его состояние представлено как **account_none**.

4.2.16. Условия консенсуса для *AccountBlocks*. Существует несколько общих условий консенсуса, налагаемых на *AccountBlock*. В частности:

- Транзакция, отображаемая как значение в расширенном словаре **transactions**, должна иметь свое значение **1t** равным его ключу.
- Все транзакции должны принадлежать аккаунту, адрес **account_addr** которого указан в *AccountBlock*.
- Если t и t' — две транзакции с $LT(t) < LT(t')$ и их ключи являются последовательными в *transactions*, то есть нет транзакции t'' с $LT(t) < LT(t') < LT(t')$, то конечное состояние t должно соответствовать начальному состоянию t' (их хэши, как явно указано в обновлениях Меркля, должны быть равны).
- Если t — транзакция с минимальным $LT(t)$, ее начальное состояние должно соответствовать начальному состоянию, указанному в **state_update** блока *AccountBlock*.
- Если t — транзакция с максимальным $LT(t)$, ее конечное состояние должно совпадать с конечным состоянием, указанным в **state_update** блока *AccountBlock*.
- Список транзакций должен быть непустым.

Эти условия просто выражают факт, что состояние аккаунта может изменяться только в результате выполнения транзакции (см. [1.2.1](#)).

4.2.17. Коллекция всех транзакций в блоке. Все транзакции в блоке представлены (см. [1.2.1](#)):

_ (HashMapAugE 256 AccountBlock Grams) = ShardAccountBlocks;

4.2.18. Условия консенсуса для коллекции всех транзакций. Снова налагаются условия консенсуса на эту структуру, требуя, чтобы значение по ключу ζ было *AccountBlock* с адресом, равным ζ . Дополнительные условия консенсуса связывают эту структуру с начальным и конечным состояниями шардчейна, указанными в блоке, требуя:

- Если *ShardAccountBlock* не имеет ключа ζ , то состояние аккаунта ζ в начальном и конечном состоянии блока должно совпадать (или оно должно отсутствовать в обоих).
- Если ζ присутствует в *ShardAccountBlock*, его начальное и конечное состояния, как указано в *AccountBlock*, должны соответствовать тем, что указаны в начальном и конечном состояниях блока шардчейна, выраженные экземплярами *ShardAccountBlocks* (см. [4.1.9](#)).

Эти условия подтверждают, что состояние шардчейна действительно состоит из состояний отдельных аккаунтов.

4.3 Описания транзакций

Этот раздел представляет специфические схемы TL-B для описания транзакций в соответствии с классификацией, описанной в [4.2.4](#).

4.3.1. Причины опущения данных из описания транзакции. Описание транзакции для блокчейна, включающего Тьюринг-полную виртуальную машину для исполнения смарт-контрактов, по своей сути является неполным. Действительно, полное описание должно было бы содержать все промежуточные состояния виртуальной машины после выполнения каждой инструкции, что невозможно поместить в блок блокчейна разумного размера. Поэтому описание транзакции, скорее всего, будет содержать только общее количество шагов и хэши начального и конечного состояний виртуальной машины. Проверка такой транзакции обязательно потребует выполнения смарт-контракта, чтобы воспроизвести все промежуточные шаги и конечный результат.

Если мы сжимаем последовательность всех промежуточных шагов виртуальной машины до хэшей начального и конечного состояний, тогда в описание транзакции вообще не нужно включать никаких деталей: валидатор, способный проверить выполнение виртуальной машины, сможет проверить все другие действия транзакции, начиная с ее исходных данных без этих деталей.

4.3.2. Причины включения данных в описание транзакции. Несмотря на вышеизложенные соображения, существует несколько причин для включения некоторых деталей в описание транзакции:

- Мы хотим наложить внешние условия консенсуса на транзакцию, так, чтобы хотя бы валидность потока значений внутри транзакции и валидность входящих и исходящих сообщений можно было быстро проверить без вызова виртуальной машины (см. [4.2.14](#)). Это по крайней мере гарантирует инвариантность общего количества каждой криптовалюты в блокчейне, даже если это не гарантирует правильность его распределения.
- Мы хотим иметь возможность отслеживать основные изменения состояния аккаунта (такие как его создание, активация или замораживание) по данным, хранящимся в описании транзакции, не выясняя пропущенные детали транзакции. Это упрощает проверку условий консенсуса между состояниями аккаунтчейна и шардчейна в блоке.
- Наконец определенная информация, такая как общее количество шагов виртуальной машины, хэши ее начального и конечного состояний, общее потребление газа и код выхода, могла бы значительно упростить отладку и реализацию программного обеспечения TON Blockchain (Эта информация помогла бы программисту понять, что произошло в конкретном блоке блокчейна).

С другой стороны, мы стремимся минимизировать размер каждой транзакции, поскольку хотим максимизировать количество транзакций, которые могут поместиться в каждом блоке (ограниченного размера). Поэтому вся информация, не требующаяся по одной из вышеуказанных причин, опускается.

4.3.3. Описание фазы хранения. Фаза хранения присутствует в нескольких видах транзакций, поэтому для этой фазы используется общее представление:

```
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. Описание фазы кредитования. Фаза кредитования может привести к получению некоторых причитающихся платежей:

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

Сумма **due_fees_collected** и **credit** должна равняться стоимости полученного сообщения + **ihr_fee**, если сообщение не было получено через IHR (в противном случае **ihr_fee** присуждается валидаторам).

4.3.5. Описание вычислительной фазы. Вычислительная фаза состоит из вызова TVM с корректными входными данными. В некоторых случаях TVM не может быть вызвана вообще (например, если аккаунт отсутствует, не инициализирован или заморожен, и обрабатываемое входящее сообщение не имеет полей кода или данных или эти поля некорректны); это отражено соответствующими конструкторами.

```
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;
```

Конструкция TL-B `_:[...]` описывает ссылку на ячейку, содержащую поля, перечисленные в квадратных скобках. Таким образом, несколько полей могут быть перенесены из ячейки, содержащей большую запись, в отдельную под ячейку.

4.3.6. Пропущенная вычислительная фаза. Если вычислительная фаза была пропущена, возможные причины включают:

- Отсутствие средств для покупки газа.
- Отсутствие состояния (т. е. кода и данных смарт-контракта) как в аккаунте (несуществующем, неинициализированном или замороженном), так и в сообщении.
- Недействительное состояние, переданное в сообщении (т. е. хэш состояния отличается от ожидаемого значения) для замороженного или неинициализированного аккаунта.

4.3.7. Валидная вычислительная фаза. Если нет причин пропускать вычислительную фазу, TVM вызывается и результаты вычисления регистрируются. Возможные параметры следующие:

- Флаг `success` устанавливается тогда и только тогда, когда `exit_code` равен 0 или 1.
- Параметр `msg_state_used` отражает, было ли использовано состояние, переданное в сообщении. Если он установлен, флаг `account_activated` отражает, привело ли это к активации ранее замороженного, неинициализированного или несуществующего аккаунта.

- Параметр **gas_fees** отражает общую сумму газа, собранную валидаторами за выполнение этой транзакции. Она должна быть равна произведению **gas_used** и **gas_price** из текущего заголовка блока.
- Параметр **gas_limit** отражает лимит газа для этого экземпляра TVM. Он равен меньшему из либо Grams, начисленных в фазе кредитования от входящего сообщения, либо цене газа, либо глобальному лимиту газа на транзакцию.
- Параметр **gas_credit** может быть ненулевым только для внешних входящих сообщений. Он равен меньшему из двух значений: количество газа, которое может быть оплачено из баланса аккаунта, или максимальный кредит газа.
- Параметры **exit_code** и **exit_args** представляют статусные значения, возвращаемые TVM.
- Параметры **vm_init_state_hash** и **vm_final_state_hash** являются хэшами начального и конечного состояний TVM, а **vm_steps** — это общее количество шагов, выполненных TVM, обычно равное двум + количество выполненных инструкций, включая неявные RET.³³

4.3.8. Описание фазы действия. Фаза действия наступает после валидной вычислительной фазы. Она пытается выполнить действия, сохраненные TVM во время вычислительной фазы в *action list*. Это может не удастся, поскольку список действий может оказаться слишком длинным, содержать недействительные действия или действия, которые не могут быть завершены (например, из-за недостатка средств для создания исходящего сообщения с требуемым значением).

```
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. Описание фазы отскока.

```
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. Описание обычной транзакции.

³³ Обратите внимание, что эта запись не отражает изменение состояния аккаунта, поскольку транзакция все еще может быть прервана во время фазы действия. В этом случае новые данные, косвенно ссылаемые по **vm_final_state_hash**, будут отброшены.

```

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;

```

Несколько условий консенсуса накладываются на эту структуру:

- **action** отсутствует тогда и только тогда, когда вычислительная фаза была неудачной.
- Флаг **aborted** устанавливается либо если фаза действий отсутствует, либо если фаза действий была неудачной.
- Фаза отскока происходит, только если установлен флаг **aborted** и входящее сообщение было способным к отскоку (bounceable).

4.3.11. Описание транзакции хранения. Транзакция хранения состоит только из одной отдельной фазы хранения:

```

trans_storage$0001 storage_ph:TrStoragePhase
= TransactionDescr;

```

4.3.12. Описание tick и tock транзакций. Транзакции tick и tock похожи на обычные транзакции без входящего сообщения, поэтому в них нет фаз кредитования или отскока:

```

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

```

4.3.13. Транзакции установки и подготовки разделения. Транзакция подготовки разделения похожа на tock-транзакцию в мастерчейне, но она должна генерировать ровно одно специальное сообщение-конструктор; в противном случае фаза действий прерывается.

```

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;

```

Обратите внимание, что транзакция установки разделения для нового родственного аккаунта ζ' относится к соответствующей транзакции подготовки разделения предыдущего существующего аккаунта ζ .

4.3.14. Транзакции установки и подготовки объединения. Транзакция подготовки объединения преобразует состояние и баланс аккаунта в сообщение, а последующая транзакция установки объединения обрабатывает это состояние:

```

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 Вызов смарт-контрактов в TVM

Этот раздел описывает точные параметры, с которыми TVM вызывается во время вычислительной фазы обычных и других транзакций.

4.4.1. Код смарт-контракта. Код смарт-контракта обычно является частью постоянного состояния аккаунта, по крайней мере, если аккаунт *активен* (см. [4.1.6](#)). Однако замороженный или *неинициализированный* (или несуществующий) аккаунт может не иметь постоянного состояния, за исключением, возможно, баланса и хэша его предполагаемого состояния (равного адресу аккаунта для неинициализированных аккаунтов). В этом случае код должен быть предоставлен в поле `init` входящего сообщения, обрабатываемого транзакцией (см. [3.1.7](#)).

4.4.2. Постоянные данные смарт-контракта. Постоянные данные смарт-контракта хранятся вместе с его кодом, и к ним применимы замечания, аналогичные сделанным выше в [4.4.1](#). В этом отношении код и постоянные данные смарт-контракта являются лишь двумя частями его постоянного состояния, которые отличаются только тем, как они обрабатываются TVM во время выполнения смарт-контракта.

4.4.3. Библиотечная среда смарт-контракта. *Библиотечная среда* смарт-контракта — это хэшмап, сопоставляющий 256-битные хэши ячеек (представления) с соответствующими ячейками. Когда внешняя ссылка на ячейку используется во время выполнения смарт-контракта, ячейка, на которую ссылаются, ищется в библиотечной среде, и внешняя ссылка на ячейку прозрачно заменяется найденной ячейкой.

Библиотечная среда для вызова смарт-контракта формируется следующим образом:

1. Глобальная библиотечная среда для воркчайна под вопросом берется из текущего состояния мастерчайна.³⁴
2. Затем она расширяется локальной библиотечной средой смарт-контракта, хранящейся в поле **library** состояния смарт-контракта. Только 256-битные ключи, равные хэшам соответствующих значений ячеек, учитываются. Если ключ присутствует в обеих библиотечных средах, локальная библиотечная среда имеет приоритет при слиянии двух библиотечных сред.
3. Наконец библиотека сообщений, хранящаяся в поле **library** поля **init** исходящего сообщения, также учитывается. Однако, если аккаунт заморожен или не инициализирован, поле **library** в сообщении считается частью предложенного состояния аккаунта и используется вместо локальной библиотечной среды на предыдущем шаге. Библиотека сообщений имеет более низкий приоритет, чем любая из глобальных библиотечных сред.

4.4.4. Начальное состояние TVM. Новый экземпляр TVM инициализируется перед выполнением смарт-контракта следующим образом:

- Исходный **cc** (текущее продолжение) инициализируется с использованием среза ячейки, созданного из ячейки **code**, содержащего код смарт-контракта, вычисленного, как описано в [4.4.1](#).
- **cp** (кодовая страница TVM) установлен в ноль. Если смарт-контракт хочет использовать другую кодовую страницу TVM *x*, он должен переключиться на нее, используя **SETCODEPAGE** *x* в качестве первой инструкции своего кода.
- Регистр управления **c0** (продолжение возврата) инициализируется экстраординарным продолжением **ec_quit** с параметром 0. При выполнении это продолжение приводит к завершению TVM с кодом выхода 0.
- Регистр управления **c1** (альтернативное продолжение возврата) инициализируется экстраординарным продолжением **ec_quit** с параметром 1. При вызове он приводит к завершению TVM с кодом выхода 1 (Обратите внимание, что завершение с кодом выхода 0 или 1 считается успешным завершением).

³⁴ Самый распространенный способ создания общих библиотек для TVM — это публикация ссылки на корневую ячейку библиотеки в мастерчайне.

- Регистр управления **c2** (обработчик исключений) инициализируется экстраординарным продолжением **ec_quit_exc**. При вызове он берет верхнее целое число из стека (равное номеру исключения) и завершает TVM с кодом выхода, равным этому целому числу. Таким образом, по умолчанию все исключения завершают выполнение смарт-контракта с кодом выхода, равным номеру исключения.
- Регистр управления **c3** (словарь кода) инициализируется ячейкой с кодом смарт-контракта, аналогично начальному текущему продолжению (**cc**).
- Регистр управления **c4** (корень постоянных данных) инициализируется постоянными данными смарт-контракта.³⁵
- Регистр управления **c5** (корень действий) инициализируется пустой ячейкой. Примитивы «выходных действий» TVM, такие как **SENDMSG**, используют **c5** для накопления списка действий (например, исходящих сообщений), которые должны быть выполнены при успешном завершении смарт-контракта (см. [4.2.7](#) и [4.2.8](#)).
- Регистр управления **c7** (корень временных данных) инициализируется единственным элементом *Tuple*, единственным компонентом которого является *Tuple*, содержащий экземпляр *SmartContractInfo* с балансом смарт-контракта и другой полезной информацией (см. [4.4.10](#)). Смарт-контракт может заменить временные данные, особенно все компоненты *Tuple* в **c7**, кроме первого, любыми другими необходимыми временными данными, которые могут потребоваться. Однако оригинальное содержимое *SmartContractInfo* в первом компоненте *Tuple*, находящегося в **c7**, проверяется и иногда модифицируется примитивами **SENDMSG** TVM и другими примитивами "выходных действий" TVM.
- Пределы газа **gas** = (g_m, g_l, g_c, g_r) инициализируются следующим образом:
 - Максимальный предел газа g_m устанавливается как меньшее из либо общего баланса смарт-контракта в Gram (после фазы кредитования, т. е. вместе со значением входящего сообщения), деленного на текущую цену газа, либо глобального предела газа за выполнение.³⁶
 - Текущий предел газа g_l устанавливается как меньшее из либо значения входящего сообщения в Gram, деленное на цену газа, либо глобального предела газа за выполнение. Таким образом, всегда $g_l \leq g_m$. Для входящих внешних сообщений $g_l = 0$, поскольку они не могут нести никакого значения.
 - Кредит газа g_c устанавливается в ноль для входящих внутренних сообщений и как меньшее из либо g_m , либо фиксированного малого значения (по умолчанию кредит газа для внешних сообщений конфигурируемый параметр) для входящих внешних сообщений.
 - Наконец оставшийся предел газа g_r автоматически инициализируется как $g_l + g_c$.

³⁵ Постоянные данные смарт-контракта не обязательно должны быть загружены полностью для выполнения. Вместо этого загружается корень, и TVM может загружать другие ячейки по их ссылкам из корня только по мере необходимости, таким образом предоставляя форму виртуальной памяти.

³⁶ И глобальный предел газа, и цена газа являются конфигурируемыми параметрами, определяемыми текущим состоянием мастерчайна.

4.4.5. Начальный стек TVM для обработки внутреннего сообщения. После инициализации TVM, как описано в [4.4.4](#), его стек инициализируется путем добавления аргументов в функцию `main()` смарт-контракта следующим образом:

- Баланс смарт-контракта b в Gram (после зачисления значения входящего сообщения) передается как *целое число*, представляющее количество нанограммов.
- Баланс входящего сообщения m b_m в Gram передается как *целое число*, представляющее количество нанограммов.
- Входящее сообщение m передается как ячейка, содержащая сериализованное значение типа *Message X*, где X — это тип тела сообщения.
- Тело $m_b : X$ входящего сообщения, равное значению поля **body** m , передается как срез ячейки.
- Наконец *селектор функции s, целое число*, обычно равное нулю, добавляется в стек.

После этого выполняется код смарт-контракта, равный его начальному значению с3. Он выбирает правильную функцию в соответствии с s , которая должна обработать оставшиеся аргументы функции и завершиться.

4.4.6. Обработка входящего внешнего сообщения. Входящее внешнее сообщение обрабатывается аналогично [4.4.4](#) и [4.4.5](#) со следующими модификациями:

- Селектор функции s устанавливается в -1, а не в 0.
- Баланс Gram входящего сообщения b_m всегда равен 0.
- Начальный текущий предел газа g_l всегда равен 0. Однако начальный кредит газа $g_c > 0$.

Смарт-контракт должен завершиться с $g_c = 0$ или $g_r \geq g_c$; в противном случае транзакция и блок, содержащий ее, недействительны. Валидаторы или коллаторы, предлагающие блок-кандидат, никогда не должны включать транзакции, обрабатывающие входящие внешние сообщения, которые недействительны.

4.4.7. Обработка tick и tock транзакций. Стек TVM для обработки tick и tock транзакций (см. [4.2.4](#)) инициализируется путем добавления следующих значений:

- Баланс Gram текущего аккаунта b в нанограммах (*целое число*).
- 256-битный адрес текущего аккаунта ζ внутри мастерчайна, представленный как беззнаковое *целое число*.
- Целое число, равное 0 для tick-транзакций и -1 для tock-транзакций.
- Селектор функции s , равный -2.

4.4.8. Обработка транзакций подготовки разделения. Для обработки транзакций подготовки разделения (см. [4.3.13](#)) стек TVM инициализируется путем добавления следующих значений:

- Баланс Gram текущего аккаунта b .

- Срез, содержащий *SplitMergeInfo* (см. [4.3.13](#)).
- 256-битный адрес текущего аккаунта ζ .
- 256-битный адрес родственного аккаунта $\tilde{\zeta}$.
- Целое число $0 \leq d \leq 63$, равное позиции единственного бита, в котором ζ и $\tilde{\zeta}$ различаются.
- Селектор функции s , равный -3.

4.4.9. Обработка транзакций установки объединения. Для обработки транзакций установки объединения (см. [4.3.14](#)) стек TVM инициализируется путем добавления следующих значений:

- Баланс Gram текущего аккаунта b (уже объединенный с балансом родственного аккаунта).
- Баланс Gram родственного аккаунта b' , взятый из входящего сообщения m .
- Сообщение m от родственного аккаунта, автоматически сгенерированное транзакцией подготовки объединения. Его поле *init* содержит конечное состояние \hat{S} родственного аккаунта.
- Состояние \hat{S} родственного аккаунта, представленное *StateInit* (см. [3.1.7](#)).
- Срез, содержащий *SplitMergeInfo* (см. [4.3.13](#)).
- 256-битный адрес текущего аккаунта ζ .
- 256-битный адрес родственного аккаунта $\tilde{\zeta}$.
- Целое число $0 \leq d \leq 63$, равное позиции единственного бита, в котором ζ и $\tilde{\zeta}$ различаются.
- Селектор функции s , равный -4.

4.4.10. Информация о смарт-контракте. Структура информации о смарт-контракте *SmartContractInfo*, передаваемая в первом компоненте *Tuple*, содержащегося в регистре управления **c7**, также является *Tuple*, содержащим следующие данные:

```
[ 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;
```

Другими словами, первый компонент этого кортежа — это *целое число magic*, всегда равное **0x076ef1ea**, второй компонент — это *целое число actions*, изначально инициализированное нулем, но увеличивающееся на единицу каждый раз, когда устанавливается выходное действие, установленное не-**RAW** примитивом вывода TVM, и так далее. Оставшийся баланс представлен парой, т. е. двухкомпонентным *Tuple*: первый компонент — это баланс в нанограммах, а второй компонент — это словарь с 32-битными ключами, представляющий все другие валюты, если таковые имеются (см. [3.1.6](#)).

Поле *rand_seed* (беззнаковое 256-битное целое число) здесь инициализируется детерминировано, начиная с *rand_seed* блока, адреса аккаунта, хэша обрабатываемого входящего сообщения (если есть) и логического времени транзакции *trans_lt*.

4.4.11. Сериализация выходных действий. Выходные действия смарт-контракта накапливаются в связанном списке, хранящемся в регистре управления **c5**. Список выходных действий сериализуется как значение типа *OutList n*, где *n* — это длина списка:

```
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 Структура блока

Эта глава представляет структуру блока, используемого в TON Blockchain, комбинируя структуры данных, описанные отдельно в предыдущих главах, чтобы создать полное описание блока шардчайна. Помимо схем TL-B, которые определяют представление блока шардчайна в виде дерева ячеек, эта глава описывает точные форматы сериализации для получающихся пакетов (коллекций) ячеек, необходимых для представления блока шардчайна в виде файла.

Блоки мастерчайна похожи на блоки шардчайна, но имеют некоторые дополнительные поля. Необходимые изменения обсуждаются отдельно в разделе [5.2](#).

5.1 Структура блока шардчайна

Этот раздел перечисляет структуры данных, которые должны содержаться в блоке и состоянии шардчайна, и завершается представлением формальной схемы TL-B блока шардчайна.

5.1.1. Компоненты состояния шардчайна. Состояние шардчайна состоит из:

- *ShardAccounts*, разделенная часть состояния шардчайна (см. [1.2.2](#)), содержащая состояние всех аккаунтов, назначенных этому шарду (см. [4.1.9](#)).
- *OutMsgQueue*, очередь исходящих сообщений шардчайна (см. [3.3.6](#)).
- *SharedLibraries*, описание всех общих библиотек шардчайна (сейчас есть только в мастерчайне).
- Логическое время и unixtime последней модификации состояния.
- Общий баланс шарда.
- Хэш-ссылка на самый последний блок мастерчайна, косвенно описывающей состояние мастерчайна и через него состояние всех других шардчайнов TON Blockchain (см. [1.5.2](#)).

5.1.2. Компоненты блока шардчайна. Блок шардчайна должен содержать:

- Список *подписей валидаторов* (см. [1.2.6](#)), который является внешним по отношению ко всем другим содержимым блока.
- *BlockHeader*, содержащий общую информацию о блоке (см. [1.2.5](#)).
- Хэш-ссылки на непосредственно предшествующий блок или блоки того же шардчайна и на самый последний блок мастерчайна.
- *InMsgDescr* и *OutMsgDescr*, дескрипторы входящих и исходящих сообщений (см. [3.2.8](#) и [3.3.5](#)).
- *ShardAccountBlocks*, коллекция всех транзакций, обработанных в блоке (см. [4.2.17](#)), вместе со всеми обновлениями состояний аккаунтов, назначенных этому шарду. Это *разделенная* часть блока шардчайна (см. [1.2.2](#)).
- *Поток значений*, описывающий общий объем импортированных значений из предыдущих блоков того же шардчайна и из входящих сообщений, общий объем

значений, экспортируемых исходящими сообщениями, общие суммы, собранные валидаторами и общее оставшееся значение в шарде.

- *Обновление Меркла* (см. [4, 3.1]) состояния шардчейна. Такое обновление Меркла содержит хэши начальных и конечных состояний шардчейна по отношению к блоку, а также все новые ячейки конечного состояния, которые были созданы при обработке блока.³⁷

5.1.3. Общие части структуры блока для всех воркчейнов. Помните, что разные воркчейны могут определять свои собственные правила обработки сообщений, другие типы транзакций, другие компоненты состояния и другие способы сериализации всех этих данных. Однако некоторые компоненты блока и его состояния должны быть общими для всех воркчейнов, чтобы поддерживать совместимость между различными воркчейнами. Такие общие компоненты включают:

- *OutMsgQueue*, очередь исходящих сообщений шардчейна, которая сканируется соседними шардчейнами для сообщений, адресованных им.
- Внешняя структура *InMsgDescr* в виде хэшмапа с 256-битными ключами, равными хэшам импортированных сообщений. (Сами дескрипторы входящих сообщений не обязательно должны иметь такую же структуру.)
- Некоторые поля в заголовке блока, идентифицирующие шардчейн и блок вместе с путями от заголовка блока к другой информации, указанной в этом списке.
- Информация о потоке значений.

5.1.4. TL-B схема для состояния шардчейна. Состояние шардчейна (см. 1.2.1 и 5.1.1) сериализуется согласно следующей TL-B схеме:

```
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;
```

³⁷ В принципе, экспериментальная версия TON Blockchain может выбрать хранение только хэшей начальных и конечных состояний шардчейна. Обновление Меркла увеличивает размер блока, но это удобно для полных узлов, которые хотят хранить и обновлять свою копию состояния шардчейна. В противном случае, полным узлам придется повторять все вычисления, содержащиеся в блоке, чтобы вычислить обновленное состояние шардчейна самостоятельно.

Поле **custom** обычно присутствует только в мастерчайне и содержит все данные, специфичные для мастерчайна. Однако другие воркчайны могут использовать ту же ячейку для ссылки на свои конкретные данные состояния.

5.1.5. Описание общих библиотек. Общие библиотеки в настоящее время могут быть только в блоках мастерчайна. Они описываются экземпляром *HashmapE(256, LibDescr)*, где 256-битный ключ является представлением хэша библиотеки, а *LibDescr* описывает одну библиотеку:

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

Здесь **publishers** — это хэшмап с ключами, равными адресам всех аккаунтов, которые опубликовали соответствующую общую библиотеку. Общая библиотека сохраняется до тех пор, пока хотя бы один аккаунт сохраняет ее в своей коллекции опубликованных библиотек.

5.1.6. TL-B схема для неподписанного блока шардчайна. Точный формат неподписанного (см. [1.2.6](#)) блока шардчайна представлен следующей TL-B схемой:

```
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;
```

Поле **custom** обычно присутствует только в мастерчейне и содержит все данные, специфичные для мастерчайна. Однако другие воркчайны могут использовать ту же ячейку для ссылки на свои конкретные данные блока.

5.1.7. Описание общего потока значений через блок. Общий поток значений через блок сериализуется согласно следующей TL-B схеме:

```
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;
```

Помните, что `_:[...]` — это конструкция TL-B, указывающая на то, что группа полей была перемещена в отдельную ячейку. Последние три поля могут быть ненулевыми только в блоках мастерчайна.

5.1.8 Подписанный блок шардчайна. Подписанный блок шардчайна — это просто неподписанный блок, дополненный набором подписей валидаторов:

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

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

Хэш сериализации `blk_serialize_hash` неподписанного блока `block` по существу представляет собой хэш конкретной сериализации блока в строку октетов (подробнее см. [5.3.12](#)). Подписи, собранные в `signatures`, представляют собой подписи Ed25519 (см. [A.3](#)), выполненные с использованием приватных ключей валидаторов для SHA256 конкатенации 256-битного представления хэша блока `block` и его 256-битного хэша сериализации `blk_serialize_hash`. 64-битные ключи в словаре `signatures` представляют собой первые 64 бита публичных ключей соответствующих валидаторов.

5.1.9 Сериализация подписанного блока. Общая процедура сериализации и подписания блока может быть описана следующим образом:

1. Неподписанный блок B генерируется, преобразуется в полный пакет ячеек (см. [5.3.2](#)) и сериализуется в строку октетов S_B .
2. Валидаторы подписывают объединенный 256-битный хэш:

$$H_B := \text{SHA256}(\text{HASH}_\infty(B) \cdot \text{HASH}_M(S_B)) \quad (18)$$

хэша представления B и хэша Меркля его сериализации S_B .

3. Подписанный блок шардчайна \tilde{B} генерируется из B и этих подписей валидаторов, как описано выше (см. [5.1.8](#)).
4. Этот подписанный блок \tilde{B} трансформируется в неполную коллекцию ячеек, которая содержит только подписи валидаторов, но сам неподписанный блок отсутствует в этой коллекции ячеек, будучи его единственной отсутствующей ячейкой.
5. Эта неполная коллекция ячеек сериализуется, и его сериализация предшествует ранее построенной сериализации неподписанного блока.

Результатом является сериализация подписанного блока в строку октетов. Она может быть распространена по сети или сохранена на диске.

5.2 Структура блока мастерчайна

Блоки мастерчайна очень похожи на блоки базового воркчайна. Этот раздел перечисляет некоторые из модификаторов, необходимых для получения описания блока мастерчайна из описания блока шардчайна, приведенного в [5.1](#).

5.2.1. Дополнительные компоненты, присутствующие в состоянии мастерчайна. В дополнение к компонентам, перечисленным в [5.1.1](#), состояние мастерчайна должно содержать:

- *ShardHashes* — Описывает текущую конфигурацию шардов и содержит хэши последних блоков соответствующих шардчайнов.
- *ShardFees* — Описывает общую сумму, собранную валидаторами каждого шардчайна.
- *ShardSplitMerge* — Описывает будущие события разделения/слияния шардов. СерIALIZУЕТСЯ как часть *ShardHashes*.
- *ConfigParams* — Описывает значения всех конфигурируемых параметров TON Blockchain.

5.2.2. Дополнительные компоненты, присутствующие в блоках мастерчайна. В дополнение к компонентам, перечисленным в [5.1.2](#), каждый блок мастерчайна должен содержать:

- *ShardHashes* — Описывает текущую конфигурацию шардов и содержит хэши последних блоков соответствующих шардчайнов. (Обратите внимание, что этот компонент также присутствует в состоянии мастерчайна.)

5.2.3. Описание *ShardHashes*. *ShardHashes* представлены в виде словаря с 32-битными *workchain_ids* в качестве ключей и «двоичными деревьями шардов», представленными TL-B типом *BinTree ShardDescr*, в качестве значений. Каждый лист этого двоичного дерева шардов содержит значение типа *ShardDescr*, которое описывает один шард, указывая порядковый номер **seq_no**, логическое время **lt** и хэш **hash** последнего (подписанного) блока соответствующего шардчайна.

```

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;

```

Поля **mc_seqno** из **fsm_split** и **fsm_merge** используются для сигнализации будущих событий разделения или слияния шардов. Блоки шардчейна, ссылающиеся на блоки мастерчейна с порядковыми номерами до, но не включая указанный в **mc_seqno**, генерируются обычным способом. Как только достигается указанный порядковый номер, должно произойти событие слияния или разделения шарда.

Обратите внимание, что сам мастерчейн исключен из *ShardHashes* (т. е. 32-битный индекс –1 отсутствует в этом словаре).

5.2.4. Описание *ShardFees*. *ShardFees* — это структура мастерчейна, используемая для отражения общей суммы, собранной валидаторами шардчейна до настоящего времени. Суммы, собранные таким образом, аккумулируются в мастерчейне путем их зачисления на специальный аккаунт, адрес которого является конфигурируемым параметром. Обычно этот аккаунт является смарт-контрактом, который выполняет распределение средств по всем валидаторам.

```

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;

```

Структура *ShardFees* похожа на *ShardHashes* (см. [5.2.3](#)), но словарь и двоичные деревья дополнены валютными значениями, равными значениям **total_validator_fees** из конечных состояний соответствующих блоков шардчейна. Значение, агрегированное в корне *ShardFees*, добавляется вместе с **total_validator_fees** состояния мастерчейна, образуя общую сумму сборов валидаторов TON Blockchain. Увеличение значения, агрегированного в корне *ShardFees* от начального до конечного состояния блока мастерчейна, отражается в **fees_imported** в потоке значений этого блока мастерчейна.

5.2.5. Описание *ConfigParams*. Напомним, что *конфигурируемые параметры* или *словарь конфигурации* представляют собой словарь **config** с 32-битными ключами, который хранится в первой ячейке ссылки на постоянные данные смарт-контракта конфигурации γ (см. [1.6](#)). Адрес γ смарт-контракта конфигурации и копия словаря конфигурации дублируются в полях **config_addr** и **config** структуры *ConfigParams*, явно включенной в состояние мастерчайна для облегчения доступа к текущим значениям конфигурируемых параметров (см. [1.6.3](#)).

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

5.2.6. Данные состояния мастерчайна. Данные, специфичные для состояния мастерчайна, собраны в *McStateExtra*, уже упомянутом в [5.1.4](#):

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

5.2.7. Данные блока мастерчайна. Аналогично, данные, специфичные для блоков мастерчайна, собраны в *McBlockExtra*:

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

5.3 Сериализация пакета ячеек

Описание, приведенное в предыдущем разделе, определяет способ, которым блок шардчайна представлен в виде дерева ячеек. Однако это дерево ячеек должно быть сериализовано в файл, подходящий для хранения на диске или передачи по сети. В этом разделе описываются стандартные способы сериализации дерева, DAG или пакет ячеек в строку октетов.

5.3.1. Преобразование дерева ячеек в пакет ячеек. Напомним, что значения произвольных (зависимых) алгебраических типов данных представлены в TON Blockchain в виде *деревьев ячеек*. Такое дерево ячеек преобразуется в ориентированный ациклический граф или *DAG* ячеек путем идентификации идентичных ячеек в дереве. После этого мы можем заменить каждую из ссылок каждой ячейки 32-байтовым представлением хэша ячейки, на которую сделана ссылка, и получить *пакет ячеек*. По соглашению корень исходного дерева ячеек является отмеченным элементом полученного пакета ячеек, так что любой, кто получает этот пакет ячеек и знает обозначенный элемент, может воссоздать оригиналный DAG ячеек, а следовательно и оригинальное дерево ячеек.

5.3.2. Полные пакеты ячеек. Допустим, что пакет ячеек является *полным*, если он содержит все ячейки, на которые ссылаются ее ячейки. Другими словами, полный пакет ячеек не имеет «неразрешенных» хэш-ссылок на ячейки за пределами пакета. В большинстве случаев нам нужно сериализовать только полные пакеты ячеек.

5.3.3. Внутренние ссылки в пакете ячеек. Допустим, что ссылка ячейки c на пакет ячеек B *внутренняя* (относительно B), если ячейка c_0 , на которую сделана эта ссылка, также принадлежит B . В противном случае ссылка называется *внешней*. Пакет ячеек полон тогда и только тогда, когда все ссылки его составляющих ячеек внутренние.

5.3.4. Присвоение индексов ячейкам из пакета ячеек. Пусть c_0, \dots, c_{n-1} будут n различными ячейками, принадлежащими пакету ячеек B . Мы можем перечислить эти ячейки в некотором порядке, а затем присвоить индексы от 0 до $n - 1$, так что ячейка c_i получает индекс i . Некоторые варианты упорядочивания ячеек:

- Упорядочение ячеек по их хэш-представлению. Тогда $\text{HASH}(c_i) < \text{HASH}(c_j)$, когда $i < j$.
- Топологический порядок: если ячейка c_i ссылается на ячейку c_j , то $i < j$. В общем случае существует более одного топологического порядка для одного и того же пакета ячеек. Существует два стандартных способа построения топологических порядков:
 - Порядок обхода в глубину: применить обход в глубину к ориентированному ациклическому графу ячеек, начиная с его корня (т. е. отмеченной ячейки), и перечислить ячейки в порядке их посещения.
 - Порядок обхода в ширину: то же самое, что и выше, но применяя обход в ширину.

Обратите внимание, что топологический порядок всегда назначает индекс 0 корневой ячейке пакета ячеек, построенного из дерева ячеек. В большинстве случаев мы выбираем использовать топологический порядок или порядок обхода в глубину, если хотим быть более конкретными.

Если ячейки перечислены в топологическом порядке, то проверка отсутствия циклических ссылок в пакет ячеек обязательна. С другой стороны, упорядочивание ячеек по их хэш-представлению упрощает проверку, что в пакете нет дубликатов ячеек.

5.3.5. Схема процесса сериализации. Процесс сериализации пакета ячеек B , состоящего из n ячеек, может быть описан следующим образом:

1. Перечислите ячейки из B в топологическом порядке: c_0, c_1, \dots, c_{n-1} . Тогда c_0 является корневой ячейкой B .
2. Выберите целое число s , такое что $n \leq 2^s$. Представьте каждую ячейку c_i как целое число октетов в стандартном виде (см. [1.1.3](#) или [\[4, 3.1.4\]](#)), используя беззнаковое s -битное целое big-endian число j вместо хэша $\text{HASH}(c_j)$ для представления внутренних ссылок на ячейку c_j (см. [5.3.6](#) ниже).
3. Конкатенируйте представления ячеек c_i , полученные таким образом, в порядке возрастания i .

4. При необходимости может быть построен индекс, состоящий из $n+1$ t -битных целых чисел L_0, \dots, L_n , где L_i — это общая длина (в октетах) представлений ячеек c_j с $j \leq i$, и целое число $t \geq 0$ выбрано так, чтобы $L_n \leq 2^t$
5. Сериализация пакета ячеек теперь состоит из магического числа, указывающего точный формат сериализации, за которым следуют целые числа $s \geq 0, t \geq 0, n \leq 2^s$, необязательный индекс, состоящий из $\lceil(n + 1)t / 8\rceil$ октетов, и L_n октетов с представлениями ячеек.
5. К сериализации может быть добавлена CRC32 для целей проверки целостности.

Если индекс включен, любая ячейка c_i в сериализованном пакете ячеек может быть легко доступна по ее индексу i без десериализации всех остальных ячеек или даже без загрузки всего сериализованного пакета ячеек в память.

5.3.6. Сериализация одной ячейки из пакета ячеек. Более конкретно, каждая отдельная ячейка $c = c_i$ сериализуется следующим образом, при условии, что s является кратным восьми (обычно $s = 8, 16, 24$ или 32):

1. Два байта дескриптора d_1 и d_2 вычисляются аналогично [4, 3.1.4] путем установки $d_1 = r + 8s + 16h + 32l$ и $d_2 = \lfloor b / 8 \rfloor + \lfloor b / 8 \rfloor$, где:
 - $0 \leq r \leq 4$ — количество ссылок на ячейки в ячейке c ; если c отсутствует в сериализованного пакета ячеек и представлена только ее хэшами, то $r = 7$.³⁸
 - $0 \leq b \leq 1023$ — количество бит данных в ячейке c .
 - $0 \leq l \leq 3$ — уровень ячейки c (см. [4, 3.1.3]).
 - $s = 1$ для экзотических ячеек и $s = 0$ для обычных ячеек.
 - $h = 1$, если хэши ячейки явно включены в сериализацию; в противном случае, $h = 0$. (Когда $r = 7$, мы всегда должны иметь $h = 1$).

Для отсутствующих ячеек (т. е. внешних ссылок) присутствует только d_1 , всегда равный $23 + 32l$.

2. Два байта d_1 и d_2 (если $r < 7$) или один байт d_1 (если $r = 7$) начинают сериализацию ячейки c .
3. Если $h = 1$, сериализация продолжается $l + 1$ 32-байтовыми верхними хэшами с (см. [4, 3.1.6]): $\text{HASH}_1(c), \dots, \text{HASH}_{l+1}(c) = \text{HASH}_\infty(c)$.

³⁸ Обратите внимание, что эти «отсутствующие ячейки» отличаются от библиотечных ссылочных и внешних ссылочных ячеек, которые являются видами экзотических ячеек (см. [4, 3.1.7]). Отсутствующие ячейки, напротив, введены только для цели сериализации неполных пакетов ячеек и никогда не обрабатываются TVM.

3. После этого сериализуются $[b / 8]$ данных в байтах, разделяя b бит данных на 8-битные группы и интерпретируя каждую группу как целое big-endian число в диапазоне от 0 до 255. Если b не делится на 8, то сначала к данным добавляется один двоичный 1 и до шести двоичных 0, чтобы сделать число бит данных, делимое на восемь.³⁹
4. Наконец r ссылок на ячейки c_{j1}, \dots, c_{jr} кодируются с помощью r s -битных целых big-endian чисел j_1, \dots, j_r .⁴⁰

5.3.7. Классификация схем сериализации для пакета ячеек. Схема сериализации для пакета ячеек должна указывать следующие параметры:

- 4-байтовое магическое число в начале сериализации.
- Количество бит s , используемое для представления индексов ячеек. Обычно s является кратным восьми (например, 8, 16, 24 или 32).
- Количество бит t , используемое для представления смещений сериализаций ячеек (см. 5.3.5). Обычно t также является кратным восьми.
- Флаг, указывающий, присутствует ли индекс со смещениями L_0, \dots, L_n всех сериализаций. Этот флаг может быть объединен с t , устанавливая $t = 0$, когда индекс отсутствует.
- Флаг, указывающий, добавлена ли CRC32-C ко всей сериализации для целей проверки целостности.

5.3.8. Поля, присутствующие в сериализации пакета ячеек. В дополнение к значениям, указанным в 5.3.7, зафиксированным выбором схемы сериализации для пакета ячеек, сериализация конкретного пакета ячеек должна указывать следующие параметры:

- Общее количество ячеек n , присутствующих в сериализации.
- Количество "корневых ячеек" $k \leq n$ в сериализации. Корневые ячейки сами по себе являются c_0, \dots, c_{k-1} . Все другие ячейки в пакете ячеек должны быть достижимы по цепочкам ссылок, начинающихся с корневых ячеек.
- Количество "отсутствующих ячеек" $l \leq n - k$, которые представляют ячейки, фактически отсутствующие в этом пакете ячеек, но на которые ссылаются из него. Сами отсутствующие ячейки представлены c_{n-l}, \dots, c_{n-1} , и только эти ячейки могут (и обычно должны) иметь $r = 7$. Полные пакеты ячеек имеют $l = 0$.
- Общая длина в байтах L_n сериализации всех ячеек. Если присутствует индекс, L_n не может храниться явно, поскольку его можно восстановить из индекса.

³⁹ Обратите внимание, что экзотические ячейки (с $s = 1$) всегда имеют $b \geq 8$, с типом ячейки, закодированным в первых восьми битах данных (см. [4, 3.1.7]).

⁴⁰ Если пакет ячеек не полный, некоторые из этих ссылок на ячейки могут ссылаться на отсутствующие ячейки c' из пакета, в таком случае специальные «отсутствующие ячейки» с $r = 7$ включены в пакет ячеек и им присвоены некоторые индексы j . Эти индексы затем используются для представления ссылок на отсутствующие ячейки.

5.3.9. Схема TL-B для сериализации пакета ячеек. Несколько конструкторов TL-B могут быть использованы для сериализации пакетов ячеек в последовательности октетов (т. е. 8-битного байта). Единственный, который в настоящее время используется для сериализации новых пакетов ячеек:

```
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;
```

Поле **cells** равно n , **roots** равно k , **absent** равно l , и **tot_cells_size** равно L_n (общий размер сериализации всех ячеек в байтах). Если индекс присутствует, параметры $s / 8$ и $t / 8$ сериализуются отдельно как **size** и **off_bytes**, соответственно, и флаг **has_idx** установлен. Сам индекс содержится в **index**, присутствующем только если **has_idx** установлен. Поле **root_list** содержит индексы (zero-based) корневых узлов пакета ячеек.

Два старых конструктора все еще поддерживаются в функциях десериализации пакета ячеек:

```
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. Хранение скомпилированного кода TVM в файлах. Обратите внимание, что вышеописанная процедура сериализации пакетов ячеек может использоваться для сериализации скомпилированных смарт-контрактов и другого кода TVM. Необходимо определить конструктор TL-B, аналогичный следующему:

```
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;
```

Затем скомпилированный смарт-контракт может быть представлен значением типа *CompiledSmartContract*, преобразованным в дерево ячеек, а затем в пакет ячеек, и сериализованным с использованием одного из конструкторов, перечисленных в [5.3.9](#). Полученная строка октетов может быть затем записана в файл с суффиксом **.tvc** («TVM smart contract»), и этот файл может использоваться для распространения скомпилированного смарт-контракта, его загрузки в приложение кошелька для развертывания в TON Blockchain и т. д.

5.3.11. Хэши Меркла для строки октетов. В некоторых случаях нам необходимо определить хэш Меркла $\text{HASH}_M(s)$ для произвольной строки октетов s длиной $|s|$. Мы делаем это следующим образом:

- Если $|s| \leq 256$ октетов, то хэш Меркла s это просто его SHA256:

$$\text{HASH}_M(s) := \text{SHA256}(s) \quad \text{if } |s| \leq 256. \quad (19)$$

- Если $|s| > 256$, пусть $n = 2^k$ будет наибольшей степенью двойки, меньшей $|s|$ (т. е. $k := \lfloor \log_2(|s| - 1) \rfloor$, $n := 2^k$). Если s' — префикс s длиной n , а s'' — суффикс s длиной $|s| - n$, так что s является конкатенацией $s'.s''$ s' и s'' , тогда мы определяем

$$\text{HASH}_M(s) := \text{SHA256}(\text{INT}_{64}(|s|). \text{HASH}_M(s'). \text{HASH}_M(s'')) \quad (20)$$

Другими словами, мы конкатенируем 64-битное представление little-endian $|s|$ и рекурсивно вычисленные хэши Меркла s' и s'' и вычисляем SHA256 результирующей строки.

Можно проверить, что $\text{HASH}_M(s) = \text{HASH}_M(t)$ для строк октетов s и t длиной меньше, чем $2^{64} - 2^{56}$ подразумевает $s = t$, если только не найдено столкновение хэшей SHA256.

5.3.12. Хэш сериализации блока. Конструкция из [5.3.11](#) применяется в частности к сериализации пакета ячеек, представляющей неподписанный блок шардчайна или мастерчайна. Валидаторы подписывают не только представление хэша неподписанного блока, но и "хэш сериализации", определенный как HASH_M сериализации неподписанного блока. Таким образом, валидаторы удостоверяют, что эта строка октетов действительно является сериализацией соответствующего блока.

Рекомендации

- [1] Daniel J. Bernstein, Curve25519: New Diffie–Hellman Speed Records (2006), in: M. Yung, Ye. Dodis, A. Kiayias et al, Public Key Cryptography, Lecture Notes in Computer Science 3958, стр. 207–228. Доступно на <https://cr.yp.to/ecdh/curve25519-20060209.pdf>.
- [2] Daniel J. Bernstein, Niels Duif, Tanja Lange et al., Highspeed high-security signatures (2012), Journal of Cryptographic Engineering 2 (2), стр. 77–89. Доступно на <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 Криптография на эллиптических кривых

Это приложение содержит формальное описание криптографии на эллиптических кривых, в настоящее время используемой в TON, особенно в TON Blockchain и TON Network.

TON использует две формы криптографии на эллиптических кривых: Ed25519 используется для криптографических подписей Шнорра, а Curve25519 используется для асимметричной криптографии. Эти кривые используются стандартным образом (как определено в оригинальных статьях [1] и [2] Д. Бернштейн и RFCs 7748 и 8032); однако некоторые детали сериализации, специфичные для TON, должны быть объяснены. Одна из уникальных адаптаций этих кривых для TON заключается в том, что TON поддерживает автоматическое преобразование ключей Ed25519 в ключи Curve25519, так что те же ключи могут использоваться как для подписей, так и для асимметричной криптографии.

A.1 Эллиптические кривые

Некоторые общие факты об эллиптических кривых над конечными полями, актуальные для криптографии на эллиптических кривых, собраны в этом разделе.

A.1.1. Конечные поля. Мы рассматриваем эллиптические кривые над конечными полями. Для целей алгоритмов Curve25519 и Ed25519 мы будем в основном заниматься эллиптическими кривыми над конечным простым полем $k := F_p$, где $p = 2^{255} - 19$ — простое число, и над конечными расширениями F_q из F_p , особенно квадратичным расширением F_{p^2} .⁴¹

A.1.2. Эллиптические кривые. Эллиптическая кривая $E = (E, O)$ над полем k является геометрически целой гладкой проективной кривой E / k рода $g = 1$ с отмеченной k -рациональной точкой $O \in E(k)$. Известно, что любая эллиптическая кривая E над полем k может быть представлена в (общей) форме Вейерштрасса:

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

Более точно, это только аффинная часть эллиптической кривой, записанной в координатах (x, y) . Для любого расширения поля K от k , $E(K)$ состоит из всех решений $(x, y) \in K^2$ уравнения (21), называемых *конечными точками* $E(K)$, вместе с точкой на бесконечности, которая является отмеченной точкой O .

⁴¹ Арифметика по модулю p для модуля p около степени двойки может быть реализована эффективно. С другой стороны, остатки по модулю $2^{255} - 19$ не представлены 255-битными целыми числами. Это объясняет это особое значение p , выбранное Д. Бернштейном.

A.1.3. Форма Вейерштрасса в однородных координатах. В однородных координатах $[X : Y : Z]$ (21) соответствует уравнению:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \quad (22)$$

Когда $Z \neq 0$, мы можем установить $x := X/Z$, $y := Y/Z$ и получить решение (x, y) для (21) (т. е. конечную точку E). С другой стороны, единственное решение (с учетом пропорциональности) для (22) с $Z = 0$ это $[0 : 1 : 0]$; это точка на бесконечности O .

A.1.4. Стандартная форма Вейерштрасса. Когда характеристика поля k не равна 2 или 3, форма Вейерштрасса уравнения (21) или (22) может быть упрощена с помощью линейных преобразований $y' := y + a_1x/2 + a_3/2$, $x' := x + a_2/3$, что приводит к установлению $a_1 = a_3 = a_2 = 0$ и получению:

$$y^2 = x^3 + a_4x + a_6 \quad (23)$$

и

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \quad (24)$$

Такое уравнение определяет эллиптическую кривую тогда и только тогда, когда кубический многочлен $P(x) := x^3 + a_4x + a_6$ не имеет кратных корней, т. е. дискриминант $D := -4a_4^3 - 27a_6^2$ не равен нулю.

A.1.5. Сложение точек на эллиптической кривой E . Пусть K будет расширением поля k и пусть $E = (E, O)$ будет любой эллиптической кривой в форме Вейерштрасса, определенной над k . Тогда любая прямая $l \subset P^2_K$ пересекает эллиптическую кривую $E(K)$ (кривая, которая является базой изменения кривой E к полю K), т. е. кривую, определенную теми же уравнениями на большем поле K ровно в трех точках P, Q, R , рассматриваемых с кратностями. Мы определяем *сложение* точек на эллиптической кривой E (или скорее добавление его K -значных точек $E(K)$) требованием, что

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

Это требование определяет уникальный коммутативный закон $[+] : E \times_k E \rightarrow E$ на точках эллиптической кривой E , имея O в качестве нейтрального элемента. Когда эллиптическая кривая E представлена в ее форме Вейерштрасса (21), можно написать явные формулы, выражающие координаты x_{P+Q} , y_{P+Q} суммы $P + Q$ двух K -значных точек $P, Q \in E(K)$ эллиптической кривой E как рациональные функции координат $x_P, y_P, x_Q, y_Q \in K$ точек P и Q и коэффициентов $a_i \in k$ из (21).

A.1.6. Карты возведения в степень. Поскольку $E(K)$ является абелевой группой, можно определить *множественные карты* или *карты возведения в степень* $[n]X$ для любой точки $X \in E(K)$ и любого целого числа $n \in \mathbb{Z}$. Если $n = 0$, тогда $[0]X = O$; если $n > 0$, тогда $[n]X = [n-1]X + X$; если $n < 0$, тогда $[n]X = -[-n]X$. Карта $[n] = [n]_E : E \rightarrow E$ для $n \neq 0$

является *изогенией*, что означает, что это непостоянный гомоморфизм для группового закона E :

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

В частности, $[-1]_E : E \rightarrow E$, $P \mapsto -P$, является инволютивным автоморфизмом эллиптической кривой E . Если E представлена в форме Вейерштрасса, $[-1]_E$ отображает $(x, y) \mapsto (x, -y)$, и две точки $P, Q \in E(\mathbb{F}_q)$ имеют равные x -координаты тогда и только тогда, когда $Q = \pm P$.

A.1.7. Порядок группы рациональных точек E . Пусть E будет эллиптической кривой, определенной над конечным базовым полем k и пусть $K = \mathbb{F}_q$ будет конечным расширением k . Тогда $E(\mathbb{F}_q)$ является конечной абелевой группой. Известным результатом Хассе порядок n этой группы не слишком удален от q :

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

Мы будем в основном заинтересованы в случае, когда $K = k = \mathbb{F}_p$, где $q = p$ — простое число.

A.1.8. Циклические подгруппы большого простого порядка. Криптография на эллиптических кривых обычно использует эллиптические кривые, которые допускают (обязательно циклическую) подгруппу $C \subset E(\mathbb{F}_q)$ простого порядка ℓ . Эквивалентно рациональная точка $G \in E(\mathbb{F}_q)$ простого порядка ℓ может быть дана; тогда C может быть восстановлена как циклическая подгруппа $\langle G \rangle$, сгенерированная G . Чтобы проверить, что точка $G \in E(\mathbb{F}_q)$ порождает циклическую группу простого порядка ℓ , можно проверить, что $G \neq O$, но $[\ell]G = O$.

По теореме Лежандра ℓ обязательно является делителем порядка $n = |E(\mathbb{F}_q)|$ конечной абелевой группы $E(\mathbb{F}_q)$:

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

Целое число c называется *кофактором*; обычно нужно, чтобы кофактор был как можно меньше, чтобы сделать $\ell = n / c$ как можно больше. Напомним, что n всегда имеет тот же порядок величины, что и q по (27), поэтому его нельзя сильно изменить, изменения E , когда q зафиксировано.

A.1.9. Данные для криптографии на эллиптических кривых. Чтобы определить специфику криптографии на эллиптических кривых, необходимо задать базовое поле конечных элементов \mathbb{F}_q (если $q = p$ является простым, это достаточно для задания p), эллиптическую кривую E / \mathbb{F}_q (обычно представленную коэффициентами ее формы Вейерштрасса (23) или (21)), базовую точку O (которая обычно является точкой на бесконечности эллиптической кривой, записанной в форме Вейерштрасса), и генератор $G \in E(\mathbb{F}_q)$ (обычно определяемый его координатами (x, y) относительно уравнения эллиптической кривой) циклической подгруппы большого простого порядка ℓ . Простое

число ℓ и кофактор c также обычно являются частью данных криптографии на эллиптических кривых.

A.1.10. Основные операции криптографии на эллиптических кривых. Криптография на эллиптических кривых обычно взаимодействует с фиксированной циклической подгруппой C большого простого порядка ℓ внутри группы точек эллиптической кривой E над конечным полем F_q . Генератор G подгруппы C обычно фиксирован. Обычно предполагается, что, учитывая точку X из C , невозможно найти ее "дискретный логарифм относительно G " (т. е. некоторое число n по модулю ℓ такое, что $X = [n]G$) быстрее, чем за $O(\sqrt{\ell})$ операций. Наиболее важные операции, используемые в криптографии на эллиптических кривых, включают сложение точек из $C \subset E(F_q)$ и вычисление их степеней или кратных.

A.1.11. Приватные и публичные ключи для криптографии на эллиптических кривых. Обычно приватный ключ для криптографии на эллиптических кривых, описываемый данными, перечисленными в [A.1.9](#), является "случайным" целым числом $0 < a < \ell$, называемым *секретным показателем*, а соответствующий публичный ключ — это точка $A := [a]G$ (или просто ее координата x_A), подходящим образом сериализованная.

A.1.12. Кривые Монтгомери. Эллиптические кривые с конкретной формой Вейерштрасса

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

называются *кривыми Монтгомери*. Они обладают удобным свойством, что $x_{P+Q}x_{P-Q}$ может быть выражено как простая рациональная функция от x_P и x_Q :

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

Это означает, что x_{P+Q} может быть вычислен, если известны x_{P-Q} , x_P и x_Q . В частности, если известны x_P , $x_{[n]P}$ и $x_{[n+1]P}$, то могут быть вычислены $x_{[2n]P}$ и $x_{[2n+1]P}$ и $x_{[2n+2]P}$. С помощью бинарного представления $0 < n < 2^s$ можно вычислить $x_{[[n / 2^{s-i}]]P}$, $x_{[[n / 2^{s-i}]+1]P}$ для $i = 0, 1, \dots, s$, таким образом получая одну из возможных формул для $x_{[n]P}$ (этот алгоритм для быстрого вычисления $x_{[n]P}$, начиная с x_P на кривых Монтгомери, называется *лестницей Монтгомери*). Следовательно, мы видим важность кривых Монтгомери для криптографии на эллиптических кривых.

A.2 Криптография Curve25519

Этот раздел описывает хорошо известную криптографию Curve25519, предложенную Дэниелом Бернштейном [\[1\]](#) и ее использование в TON.

A.2.1. Curve25519. *Curve25519* определяется как кривая Монтгомери по формуле:

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

Порядок этой кривой составляет 8ℓ , где ℓ — простое число и $c = 8$ является кофактором. Циклическая подгруппа порядка ℓ порождается точкой G с $x_G = 9$ (это определяет G до знака y_G , который несуществен). Порядок квадратичного изгиба $2y^2 = x^3 + Ax^2 + x$ *Curve25519* составляет $4\ell'$ для другого простого числа ℓ' .⁴²

A.2.2. Параметры Curve25519. Параметры *Curve25519* следующие:

- Базовое поле: Простое конечное поле \mathbb{F}_p для $p = 2^{255} - 19$.
- Уравнение: $y^2 = x^3 + Ax^2 + x$, где $A = 486662$.
- Базовая точка G : Характеризуется $x_G = 9$ (девять — это наименьшее положительное целое число координата x генератора подгруппы большого простого порядка из $E(\mathbb{F}_p)$).
- Порядок $E(\mathbb{F}_p)$:

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

$$\ell = 2^{252} + 277423177773723535851937790883648493 \quad \text{is prime.} \quad (33)$$

- Порядок $\tilde{E}(\mathbb{F}_p)$, где \tilde{E} является квадратичным изгибом E :

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

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

A.2.3. Приватные и публичные ключи для стандартной криптографии Curve25519.

Приватный ключ для криптографии *Curve25519* обычно определяется как *секретный показатель* a , а соответствующий публичный ключ это x_A x -координата точки $A := [a]G$. Это обычно достаточно для выполнения ECDH (обмен ключами Диффи-Хеллмана на эллиптических кривых) и асимметричной криптографии на эллиптических кривых следующим образом:

⁴² Фактически, Д. Бернштейн выбрал $A = 486662$, потому что это наименьшее положительное целое число $A \equiv 2 \pmod{4}$, соответствующее модификации Монтгомери (31) над F_p для $p = 2^{255} - 19$, и тем самым квадратичный изгиб этой кривой имеет малые кофакторы. Такое расположение избегает необходимости проверять, определяет ли x -координата $x_P \in F_p$ точки P точку $(x_P, y) \in F_p^2$, лежащую на самой кривой Монтгомери или на ее квадратичном изгибе.

Если сторона хочет отправить сообщение M другой стороне, которая имеет публичный ключ x_A (и приватный ключ a), выполняются следующие вычисления. Генерируется одноразовый секретный показатель b , и вычисляются $x_B := x_{[b]G}$ и $x_{[b]A}$ с использованием метода лестницы Монтгомери. Затем сообщение M шифруется симметричным шифром, например AES, используя 256-битный "общий секрет" $S := x_{[b]A}$ в качестве ключа, и 256-битное целое число (одноразовый публичный ключ) x_B добавляется к зашифрованному сообщению. После получения сообщения сторона с публичным ключом x_A может вычислить $x_{[a]B}$ начиная с x_B (переданного с зашифрованным сообщением) и приватного ключа a . Поскольку $x_{[a]B} = x_{[b]A}$, получающая сторона восстанавливает общий секрет S и может расшифровать остаток сообщения.

A.2.4. Публичные и приватные ключи для криптографии TON Curve25519. TON использует другую форму публичных и приватных ключей криптографии Curve25519, заимствованную у криптографии Ed25519.

Приватный ключ для криптографии TON Curve25519 — это просто случайная 256-битная строка k . Она используется путем вычисления $\text{SHA512}(k)$, принимая первые 256 бит результата, интерпретируемых как 256-битное целое число little-endian a , очищая биты 0, 1, 2 и 255 в a и устанавливая бит 254, чтобы получить значение $2^{254} \leq a < 2^{255}$, делимое на восемь. Значение a таким образом полученное является секретным показателем, соответствующим k ; тем временем, оставшиеся 256 бит $\text{SHA512}(k)$ составляют *секретную соль* k'' .

Публичный ключ, соответствующий k или секретному показателю a , — это просто x -координата x_A точки $A := [a]G$. Как только a и x_A вычислены, они используются точно так же, как в A.2.3. В частности, если x_A нужно сериализовать, он сериализуется в 32 октета как беззнаковое целое little-endian число 256-бит.

A.2.5. Curve25519 в TON Network. Обратите внимание, что асимметричная криптография Curve25519, описанная в A.2.4, широко используется в TON Network, особенно протоколом ADNL (Abstract Datagram Network Layer). Однако TON Blockchain нуждается в криптографии на эллиптических кривых в основном для подписей. Для этой цели используются подписи Ed25519, описанные в следующем разделе.

A.3 Ed25519 криптография

Криптография Ed25519 широко используется для быстрых криптографических подписей как в TON Blockchain, так и в TON Network. Этот раздел описывает вариант криптографии Ed25519, используемый TON. Важное отличие от стандартных подходов (как определено Д. Бернштейном и другими в [2]) заключается в том, что TON обеспечивает автоматическое преобразование приватных и публичных ключей Ed25519 в ключи Curve25519, так что те же ключи могут использоваться как для шифрования/дешифрования, так и для подписания сообщений.

A.3.1. Скрученные кривые Эдвардса. Скрученная крива Эдвардса $E_{a,d}$ с параметрами $a \neq 0$ и $d \neq 0$ над полем k задается уравнением:

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

Если $a = 1$, это уравнение определяет (не скрученную) кривую Эдвардса. Точка $O(0, 1)$ обычно выбирается в качестве отмеченной точки $E_{a,d}$.

A.3.2. Скрученные кривые Эдвардса бирационально эквивалентны кривым

Монтгомери. Скрученная кривая Эдвардса $E_{a,d}$ бирационально эквивалентна эллиптической кривой Монтгомери

$$M_A : v^2 = u^3 + Au^2 + u \quad (37)$$

где $A = 2(a+d)/(a-d)$ и $d/a = (A-2)/(A+2)$. Бирациональная эквивалентность $\phi : E_{a,d} \rightarrow M_A$ и ее обратная ϕ^{-1} задаются следующими преобразованиями:

$$\phi : (x, y) \mapsto \left(\frac{1+y}{1-y}, \frac{c(1+y)}{x(1-y)} \right) \quad (38)$$

и

$$\phi^{-1} : (u, v) \mapsto \left(\frac{cu}{v}, \frac{u-1}{u+1} \right) \quad (39)$$

где

$$c = \sqrt{\frac{A+2}{a}} \quad (40)$$

Обратите внимание, что ϕ преобразует отмеченную точку $O(0, 1)$ из $E_{a,d}$ в отмеченную точку M_A (т. е. ее точку в бесконечности).

A.3.3. Сложение точек на скрученной кривой Эдвардса. Поскольку $E_{a,d}$ бирационально эквивалентна эллиптической кривой M_A , добавление точек на M_A может быть перенесено на $E_{a,d}$, установив

$$P + Q := \phi^{-1}(\phi(P) + \phi(Q)) \quad \text{for any } P, Q \in E_{a,d}(k). \quad (41)$$

Заметим, что отмеченная точка $O(0,1)$ является нейтральным элементом по отношению к этому сложению и что $-(x_P, y_P) = (-x_P, y_P)$.

A.3.4. Формулы для добавления точек на скрученной кривой Эдвардса. Координаты x_{P+Q} и y_{P+Q} допускают простые выражения как рациональные функции x_P, y_P, x_Q, y_Q :

$$x_{P+Q} = \frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q} \quad (42)$$

$$y_{P+Q} = \frac{y_P y_Q - a x_P x_Q}{1 - d x_P x_Q y_P y_Q} \quad (43)$$

Эти выражения могут быть эффективно вычислены, особенно если $a = -1$. Это одна из причин, почему скрученные кривые Эдвардса важны для быстрой криптографии на эллиптических кривых.

A.3.5. Скрученная кривая Эдвардса Ed25519. Ed25519 — это скрученная кривая Эдвардса $E_{-1, d}$ над \mathbb{F}_p , где $p = 2^{255} - 19$ — то же самое простое число, что используется для Curve25519, и $d = -(A - 2) / (A + 2) = -121665 / 121666$, где $A = 486662$ такое же, как в уравнении (31):

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2 \quad \text{for } x, y \in \mathbb{F}_p, p = 2^{255} - 19. \quad (44)$$

Таким образом, кривая Ed25519 $E_{-1, d}$ бирационально эквивалентна Curve25519 (31), и можно использовать $E_{-1, d}$ и формулы (42)–(43) для сложения точек либо на Ed25519, либо на Curve25519, используя (38) и (39) для преобразования точек на Ed25519 в соответствующие точки на Curve25519, и наоборот.

A.3.6. Генератор Ed25519. Генератором Ed25519 является точка G' с $y(G') = 4 / 5$ и $0 \leq x(G') < p$ — четное. Согласно (38) она соответствует точке (u, v) на Curve25519 с $u = (1 + 4 / 5) / (1 - 4 / 5) = 9$ (т. е. генератору G Curve25519, выбранному в A.2.2). В частности, $G = \phi(G')$ и G' порождает циклическую подгруппу того же большого простого порядка ℓ , что и указано в (32), и для любого целого числа a ,

$$\phi([a]G') = [a]G \quad . \quad (45)$$

Таким образом, мы можем выполнять вычисления с Curve25519 и ее генератором G или с Ed25519 и генератором G' и получать по сути те же результаты.

A.3.7. Стандартное представление точек на Ed25519. Точка $P(x, y)$ на Ed25519 может быть представлена двумя ее координатами x_P и y_P , остатками по модулю $p = 2^{255} - 19$. В свою очередь, обе эти координаты могут быть представлены беззнаковыми 255-битными или 256-битными целыми числами $0 \leq x_P, y_P < p < 2^{255}$.

Однако более компактное представление P одним беззнаковым 256-битным целым little-endian числом \tilde{P} широко используется (и используется также TON). В частности, 255 нижних битов \tilde{P} содержат y_P , $0 \leq y_P < p < 2^{255}$, и бит 255 используется для хранения $x_P \bmod 2$, нижний бит x_P . Поскольку y_P всегда можно отличить по их нижнему биту, p является нечетным.

Если достаточно знать $\pm P$ по знаку, можно не учитывать $x_P \bmod 2$ и рассматривать только 255-битное целое little-endian число y_P , произвольно устанавливая бит 255, игнорируя его ранее определенное значение или очищая его.

A.3.8. Приватный ключ для Ed25519. Приватный ключ для Ed25519 — это просто произвольная 256-битная строка k . Секретный показатель a и секретная соль k'' выводятся из k , сначала вычисляя $\text{SHA512}(k)$, а затем беря первые 256 бит этого SHA512 как представление little-endian a (с битами 255, 2, 1, 0 очищенными, а бит 254 установленным); последние 256 бит $\text{SHA512}(k)$ затем составляют k'' .

Это по сути та же процедура, что описана в [A.2.4](#), но с заменой Curve25519 на бирационально эквивалентную кривую Ed25519. (На самом деле, все наоборот: эта процедура является стандартной для криптографии на основе Ed25519, а TON расширяет эту процедуру до Curve25519.)

A.3.9. Публичный ключ для Ed25519. Публичный ключ, соответствующий приватному ключу k для Ed25519, является стандартным представлением (см. [A.3.7](#)) точки $A = [a]G'$, где a — секретный показатель (см. [A.3.8](#)), определенный приватным ключом k .

Обратите внимание, что $\phi(A)$ является публичным ключом для Curve25519, определенным тем же самым приватным ключом k согласно [A.2.4](#) и [\(45\)](#). Таким образом, мы можем преобразовать публичные ключи для Ed25519 в соответствующие публичные ключи для Curve25519 и наоборот. Приватные ключи не требуют преобразования.

A.3.10. Криптографические подписи Ed25519. Если сообщение (строка октетов) M должно быть подписано приватным ключом k , определяющим секретный показатель a и секретную соль k'' , выполняются следующие вычисления:

- $r := \text{SHA512}(k'' | M)$, интерпретируется как 512-битное целое число малого порядка байтов. Здесь $s|t$ обозначает конкатенацию строк октетов s и t .
- $R := [r]G'$ является точкой на Ed25519.
- \tilde{R} является стандартным представлением (см. [A.3.7](#)) точки R как строки из 32 октетов.
- $s := r + a \cdot \text{SHA512}(\tilde{R} | A | M) \bmod \ell$, закодировано как 256-битное целое число малого порядка байтов. Здесь \tilde{A} является стандартным представлением публичного ключа $A = [a]G'$, соответствующего k .

Подпись (Schnorr) является строкой из 64 октетов (\tilde{R}, s) , состоящей из стандартного представления точки R и 256-битного целого числа s .

A.3.11. Проверка подписей Ed25519. Для проверки подписи (\tilde{R}, s) сообщения M , предположительно сделанной владельцем приватного ключа k , соответствующего известному публичному ключу A , выполняются следующие шаги:

- Вычисляются точки $[s]G'$ и $R + [\text{SHA512}(\tilde{R} | A | M)]A$ на Ed25519.
- Если эти две точки совпадают, подпись верна.