Руководство разработчика программного обеспечения “Блокчейн-платформа IZZZIO версия 1.2.0”

ООО “Изио”
Москва, 2019

Содержание документа:

1. Введение

Данный документ содержит информацию для технических специалистов и является руководством для разработки проектов на базе блокчейн-платформы IZZZIO версия 1.2.0.

Блокчейн-платформа IZZZIO версия 1.2.0 (далее – блокчейн-платформа IZZZIO, Платформа IZZZIO, Система, ПО) - это программный комплекс, предназначенный для хранения, передачи и обработки любых данных, а также, для автоматизации бизнес-процессов с помощью применения технологии распределенного реестра.

Блокчейн-платформа IZZZIO - это универсальная блокчейн платформа для любых задач. В основе платформы лежит идея создания блокчейн экосистемы, максимально приближенной к потребностям бизнеса, облегчающая интеграцию и использование технологии блокчейн в бизнес-процессах компаний.

Исходный код платформы открыт и доступен по лицензии Apache-2.0. Использование частей или всего кода платформы в любом из проектов требует публикации исходного кода этих проектов любым доступным способом.

Основа блокчейн-платформы IZZZIO - система смарт-контрактов, выполняющихся в специальной среде выполнения EcmaContracts, которая использует JavaScript движок V8 для создания виртуализированного окружения исполнения кода контрактов. Это позволяет полностью изолировать произвольный код контрактов как от системы, так и от других контрактов, а также контролировать доступность ресурсов (ОЗУ, процессорное время, лимиты вызовов) для каждой сущности виртуальной машины.

EcmaContracts реализует принцип управления состоянием (state) сущностей контракта, при котором в цепочку блокчейн сохраняется только информация о вызовах методов смарт-контрактов и их аргументах. Результаты работы смарт контрактов сохраняются локально на ноде пользователя, и загружаются при следующем вызове контракта. Все ноды в сети одновременно повторяют каждую транзакцию цепочки с самого первого блока, до последнего существующего, что приводит к полной синхронизации состояний контрактов на каждой из нод.

Блокчейн-платформа IZZZIO - это одномерный блокчейн, устроенный по принципу 1 транзакция - 1 блок. По умолчанию в сети отсутствуют какие-либо специальные генераторы блоков, генерация блока происходит на той ноде, которой необходимо добавить блок в сеть. Также возможна конфигурация этого поведения.

Техническая сводка блокчейн-платформы IZZZIO:

Способ подтверждения блоков	**Встроены:** Своя разработка, основанная на гибридном PoW + PoA способе - Limited Confidence Proof of Activity с привязкой подтверждения по времени DLCPoA - Dynamic Limited Confidence Proof of Activity Proof-of-Authority - алгоритм подтверждения, использующий заранее выпущенный список ключей **Возможен любой другой, подключаемый с помощью системы плагинов**

Программная платформа	Node.js > 8
Хеширование	Фильтрованный SHA256 / ГОСТ Р 34.11-2012 / любой вид, предоставляемый плагином
Цифровые подписи	ECDSA-SHA-256 / ГОСТ Р 34.10-2012 / любой вид, предоставляемый плагином
Криптопровайдер	Встроенный / VipNet CSP 4.2 / предоставляемые плагином
Структура основной цепи блоков	JSON блоки с произвольным содержимым
База данных	LevelDB, Sqlite3, Memory
Смарт-контракты	ECMAScript 6 (JS ES6), тьюринг полные, с расширенным функционалом, WASM, поддержка других языков с помощью плагинов
Сетевой p2p интерфейс	WebSocket Based
API	REST / PHP Class / REPL

2. Описание конфигурации блокчейн-платформы IZZZIO версия 1.2.0

Платформа предоставляет пользователям большое количество настраиваемых возможностей. Конфигурация узлов выполняется созданием файла config.json в директории, из которой происходит запуск узла IZZZIO или с помощью явного указания пути конфигурационного файла с помощью параметра запуска --config. Файл конфигурации представляется в формате JSON.

Возможные параметры конфигурационного файла в Системе:

p2pPort: 6013,	Порт p2p
httpServer: 'localhost',	Адрес привязки RPC/API и интерфейса
rpcPassword: '',	Пароль для подключения к RPC/API и интерфейсу (HTTP Basic Auth)
initialPeers: []	Массив со списком стартовых узлов
allowMultipleConnectionsFromIp: true,	Разрешить множественные p2p подключения с одного адреса. Рекомендуемые значения: False - если в сети много зацикливаний, True - если используется прокси для коннекта
maxPeers: 80,	Максимальное количество p2p подключений
upnp: { enabled: true, token: 'iz3node'	Конфигурация автоматического поиска узлов с помощью DNS запроса Включить автоматическое обнаружение узлов в сети Токен по которому узел будет искать другие узлы

}	
networkPassword: '',	"пароль" доступа к сети. Используется для создания приватных сетей
blockAcceptCount: 20,	Количеств блоков подтверждения транзакции (LC)
hearbeatInterval: 10000,	Внутренний таймер узла. Влияет на скорость повторных попыток подключений, сохранения состояний на диск и т.д.
peerExchangeInterval: 5000,	Частота обмена списками пиров
maxBlockSend: 600,	Максимальное количество отправляемых блоков. Должно быть больше blockQualityCheck (LC)
blockQualityCheck: 100,	Количество блоков "сверх", которые запрашиваются для проверки валидности цепочки (LC)
limitedConfidenceBlockZone: 288,	Зона "доверия" (LC). Цепочку ранее этой зоны менять нельзя. Должно быть больше blockQualityCheck
generateEmptyBlockDelay: 300 * 1000,	Частота создания пустых блоков при простое сети. false - для отключения

blockHashFilter: {
blockEndls: []
}
	Фильтр корректных блоков для LCPoA
	4 символа в конце блока. Сюда должен попасть совпадающий с Genesis Block хеш

genesisTiemstamp: 1492004951 * 1000,	Таймстамп 0 блока сети в миллисекундах
lcpoaVariantTime: 1,	Количество миллисекунд, требуемое на генерацию одного хеша блока (LCPoA)
validators: []	Плагины дополнительных консенсусов. ВАЖНО: Передача блоков плагинам консенсусов идёт в том-же порядке, в котором они указаны в этом параметре.
emptyBlockInterval: 10000,	Интервал проверки необходимости выпуска пустого блока
maxTransactionAtempts: 5,	Сколько попыток добавить блок предпринимается до решения отмены добавления блока в сеть
checkExternalConnectionData: false,	Проверять внешние данные на соответствие (конфигурации других узлов)
enableMessaging: false,	Разрешить использование шины сообщений (необходима для некоторых консенсусов)
recieverAddress: '123',	Внутрисетевой адрес узла сети. Используется в шине сообщений, а также протоколе StarWave
messagingMaxTTL: 3,	Максимальный предел скачков сообщения
maximumInputSize: 2 * 1024 * 1024,	Максимальный объем обрабатываемых данных от внешних узлов. Рекомендуется не больше 15 мб

allowMultipleSocketsOnBus: false,	Разрешить на подключение сокетов с разными адресами на один адрес узла
walletFile: './wallet.json',	Адрес файла кошелька
workDir: '.',	Рабочая директория
blocksDB: 'blocks',	Как хранить БД блоков: false - для хранения в ОЗУ, mem://blocks.json для хранения в ОЗУ и записи на ПЗУ при выгрузке. Строка, для хранения на диске
blocksSavingInterval: 300000,	Интервал сохранения блоков. false = для отключения автосохранения, или количество миллисекунд
appEntry: false,	Точка входа в "приложение" DApp. False - если не требуется
startMessage: false,	Сообщение, которое выводится при запуске узла

ecmaContract: {
enabled: true,
allowDebugMessages: false,
contractInstanceCacheLifetime: 10000,
}
	Конфигурация смарт контрактов EcmaContracts
	Система обработки контрактов включена
	Разрешает вывод сообщений смарт контрактам (используется для отладки)
	Время жизни экземпляра виртуальной машины контракта

masterContract: 5	Главный контракт в системе. Должен поддерживать методы управления ресурсами
hashFunction: 'SHA256',	Функция, используемая для вычисления хэша. Регистрируется плагинами
signFunction: '',	Функция вычисления цифровой подписи и генерации паролей(пусто-по умолчанию), варианты: 'GOST' 'GOST256' 'NEWRSA'
keyLength: 2048,	Длина ключа (передается в некоторые из плагинов)
plugins: []	Массив со списком загружаемых плагинов (кроме плагинов консенсусов)

3. Язык смарт-контрактов

Система выполнения смарт-контрактов IZZZIO позволяет писать контракты с помощью языка JavaScript ES6. Смарт контракты являются полными по тьюрингу и имеют широкий функционал возможностей. При разработке доступно большинство стандартных методов и классов.

- Заменён метод Math.random(). Добавлена поддержка сохранения состояния.
- Заменены методы класса Date для поддержки сохранения состояния.

При написании контрактов необходимо использовать несколько принципов, обеспечивающих безопасность и корректность выполнения кода:

1. Необходимо писать код в синхронном стиле. Асинхронность не поддерживается.
2. Имена всех приватных методов и свойств должны начинается со знака «_» для предотвращения несанкционированного обращения к ним
3. Не использовать стандартный конструктор класса. Для инициализации класса необходимо использовать метод **init** и **deploy** для обработки первого запуска контракта
4. Для сохранения данных между запусками, необходимо использовать сохраняемые объекты KeyValue (см.п.4.1.1.1), TypedKeyValue (см.п.4.1.1.2), BlockchainArray (см.п.4.1.1.4), BlockchainMap (см.п.4.1.1.6).

Рекомендации написания безопасных смарт контрактов:

Эти рекомендации позволят избежать большинство ошибок в проектировании и написании контракта. Запущенный контракт в последующем невозможно модифицировать или обновить!

1. Для реализации токена использовать тип данных TokenRegister (см.п.4.1.1.6) или наследовать контракт от TokenContract (см.п.4.1.3.2.)
2. Для работы с числами, для которых важна точность вычислений, использовать класс BigNumber (см.п.4.2.1).
3. Необходимо наследовать любой контракт от класса Contract (см.п.4.1.3.1), который реализует основные системы защиты и обработки данных для контрактов
4. Выполнять проверку всех входных данных на типы, а также обязательное приведение данных к нужному типу
5. Выполнять проверки окружения выполнения контракта с помощью assert (см.п.4.1.5.1) и других методов исключений
6. Важные данные, такие как адрес владельца контракта, следует определять заранее в константе перед классом контракта
7. Помните, что данные, записанные в блокчейн являются открытыми, и доступны всем.

Советы по оптимизации:

1. Минимизировать чтение и запись данных в сохраняемые объекты KeyValue (см.п.4.1.1.1), TypedKeyValue (см.п.4.1.1.2), BlockchainArray (см.п.4.1.1.4), BlockchainMap (см.п.4.1.1.6)
2. Минимизировать или не использовать циклы с перебором неопределенного или очень большого количества параметров

3.1. Простейший токен

Блокчейн IZZZIO использует свой стандарт описания контрактов-токенов, схожий со стандартом ERC20 блокчейна Ethereum, однако с некоторыми отличиями. Стандарт IZ3 Token (IZ3T) подразумевает наличие определенных методов и свойств у контракта. Большинство этих методов уже реализовано во встроенном классе TokenContract (см.п.4.1.3.2.), от которого рекомендуется наследовать любой токен-контракт.

Пример простейшего токена, соответствующему стандарту IZ3T и рекомендациям безопасности:

```
//Определение размера эмиссии
const EMISSION = 10000;

//Определение адреса владельца контракта
const CONTRACT_OWNER = 'SOME_ADDR';
```

```
class TestToken extends TokenContract{

   //Инициализация токена. Обратите внимание, что init используется в качестве конструктора
   init() {
      super.init(EMISSION);
   }

   //Свойство данных о контракте
   get contract() {
      return {
         name: 'Token name',
         ticker: 'TokenTicker',
         owner: CONTRACT_OWNER,
         emission: EMISSION,
         type: 'token',
      };
   }
}

//Передача класса контракта в управление виртуальной машине
global.registerContract(TestToken);
```

Контракт наследует основные методы стандарта от класса TokenContract. При запуске контракта эмиссия токенов будет произведена на адрес CONTRACT_OWNER. Учет количества и движения токенов производится внутри TokenRegister объявленного в TokenContract

3.2. Запуск контракта

3.2.1. Запуск с помощью DApp

Одним из способов запуска контракта в сеть является запуск через DApp (см.п.6) SDK платформы IZZZIO.

Код DApp приложения для запуска контракта:

```
const CONTRACT_CODE = ''; //Переменная, содержащая код контракта

const RESOURCES = 1; //Количество токенов выделяемых для аренды ресурсов

//Модуль DApp SDK IZZZIO
const DApp = require(global.PATH.mainDir + '/app/DApp');

class App extends DApp {
      init() {
         that = this;
         that.contracts.ecmaContract.deployContract(CONTRACT_CODE, RESOURCES, function (deployedContract) {
               console.log('Contract deployed at address', deployedContract.address);
         });
      };
}
```

Для запуска DApp необходимо добавить в config.json параметр appEntry и указать путь до точки входа в приложение. Например «appEntry»: «./BigNet/startNetwork.js», из файла BigNet/configStart.json в репозитории.

После запуска DApp контракт будет запущен в сеть, и его адрес будет выведен на экран.

3.2.2. Запуск с помощью API

Для запуска контракта с помощью API используется метод
contracts/ecma/deployContract

POST

- **resourceRent** - Число. Количество токенов выделяемых для аренды ресурсов
- **source** - Строка. Исходный JavaScript код смарт контракта

См. также: Block (п.7.4).

3.3. Взаимодействие с контрактом

3.3.1. Взаимодействие из DApp

Для взаимодействия в сети IZZZIO рекомендуется использовать методы ecmaContracts из DApp SDK. Например, для получения общего количества существующих токенов контракта-токена можно использовать метод:

```
let supply = await that.contracts.ecmaPromise.callMethodRollback(CONTRACT_ADDRESS, 'totalSupply', [], {});
```

Вызов этого метода запускает метод totalSupply из контракта. После завершения выполнения состояние контракта откатывается до состояния до запуска, и возвращается значение, которое вернул метод totalSupply, т.е. общее количество существующих токенов.

Для взаимодействия с некоторыми стандартными форматами контрактов существуют встроенные классы-обёртки, облегчающие вызовы методов и получения свойств. Например для IZ3 Token класс-обертка - DApp-TokenContractConnector (см.п.6.4.2). Использовать его можно так:

```
let token = new TokenContractConnector(that.ecmaContract, CONTRACT_ADDRESS);

//Аналог примера выше
let supply =  await token.totalSupply();

//Выполнение транзакции на 10 токенов
await token.transfer(SOME_ADDRESS,'10');
```

Подробнее про DApp-TokenContractConnector вы сможете прочитать в соответствующем разделе.

3.3.2. Запуск с помощью API

Для взаимодействия с контрактом с помощью API используются методы:

- **GET** contracts/ecma/getContractInfo/*:contractAddress:*
- **GET** contracts/ecma/getContractProperty/*:contractAddress:*/*:property:*
- **POST** contracts/ecma/callMethod/*:contractAddress:*/*:method:*

- **POST** contracts/ecma/deployMethod/*:contractAddress:*/*:method:*

Пример запроса totalSupply контракта:
POST
contracts/ecma/callMethod/*1*/*totalSupply*

- argsEncoded = ‘{}’ - Строка. Сериализованное значение со списком аргумента вызова

При выполнении этого вызова будет выполнен:

1. Запуск смарт контракта
2. Вызов метода totalSupply
3. Остановка контракта
4. Откат изменений выполненных во время работы
5. Отправка результата в ответ на запрос API метода

Подробное описание API смотрите в разделе API (п.4.3).

4. EcmaContracts

EcmaContracts реализует среду управления и выполнения смарт контрактов. EcmaContracts контролирует состояние, запущенные сущности виртуальных машин, потребление ресурсов, и обеспечивает среды выполнения контрактов необходимыми для работы методами и объектами. Взаимодействие с такими объектами позволяет мощные и функциональные децентрализованные системы.

Доступные объекты

- Объекты-хранилища
- События
- Объекты контрактов
- Коннекторы
- Системные классы

Другие доступные объекты

- BigNumber
- Date
- Math

API

4.1. Доступные объекты

4.1.1. Объекты-хранилища

Объекты, реализующие перенос и сохранение состояния между вызовами методов смарт контракта. Необходимы для сохранения любой информации (состояние счетов пользователя, балансы и другие)

4.1.1.1. KeyValue

Единственный реальный сохраняемый объект внутри виртуальной машины. Другие сохраняемые объекты зависят от объекта KeyValue, и по сути являются обертками на его базе. KeyValue - объект, реализующий хранилище ключ-значение. Максимальное количество записей - не ограничено. Максимальная длина ключа и записи - 512 кБ.

new KeyValue *(название хранилища)*
Аргументы:

- название хранилища *(string)* – Название хранилища для загрузки и записи данных

get *(ключ)*
Аргументы:

- ключ *(string)* – Ключ по которому будет выполняться поиск значения в хранилище

Результат:

Значение. Если не найдено - false

put *(ключ, значение)*
Аргументы:

- ключ *(string)* – Ключ по которому будет выполнена запись в хранилище
- значение *(string)* – Строка для записи в хранилище

Пример инициализации и использования методов:

```
class Test extends Contract {
  init(){
    super.init();
    //Создаём/загружаем хранилище
    this._db = new KeyValue('name');
  }

  putKeyValue(){
    this._db.put('key','value');
  }

  getKeyValue(){
    console.log(this._db.get('key')); //Выведет "value"
  }
}
```

4.1.1.2. TypedKeyValue

Аналог KeyValue (п.4.1.1.1), сохраняющий типы значений, при условии, что их можно сериализовать.

new TypedKeyValue *(название хранилища)*
Аргументы:

- название хранилища *(string)* – Название хранилища для загрузки и записи данных

get (ключ)
Аргументы:

- ключ *(string)* – Ключ по которому будет выполняться поиск значения в хранилище

Результат:

Значение. Если не найдено - false

put *(ключ, значение)*
Аргументы:

- ключ *(string)* – Ключ по которому будет выполнена запись в хранилище
- значение *(*)* – Строка для записи в хранилище

Пример инициализации и использования методов:

```
class Test extends Contract {
    init(){
        super.init();
        //Создаём/загружаем хранилище
        this._db = new KeyValue('name');
    }

    putKeyValue(){
        this._db.put('number',1337);
        this._db.put('string',"Hello");
    }

    getKeyValue(){
        console.log(typeof this._db.get('number')); //Выведет "number"
        console.log(typeof this._db.get('string')); //Выведет "string"
    }
}
```

4.1.1.3. BlockchainArraySafe

Массиво-подобная структура на основе KeyValue (п.4.1.1.1). Не содержит методы, которые могут привести к проблеме переполнения ОЗУ (join(), slice(), splice() и другие).

new BlockchainArraySafe *(название хранилища)*
Аргументы:

- название хранилища *(string)* – Название хранилища для загрузки и записи данных

```
class Test extends Contract {
    init(){
        super.init();
        //Создаём/загружаем хранилище
        this._db = new BlockchainArraySafe('name');
    }

    doSomething(){
        this._db[0]=1;
    }

    doSomethingNext(){
        this._db[0]++;
    }

    test(){
        this.doSomething();
        this.doSomethingNext();
        console.log(this._db[0]); //Выведет "2"
    }
}
```

4.1.1.4. BlockchainArray

Данный объект является расширением BlockchainArraySafe (п.4.1.1.3). Содержит полифилы методов, которые могут привести к переполнению доступной памяти. Рекомендуется использовать только на небольших массивах данных.

new BlockchainArray *(название хранилища)*
Аргументы:

- название хранилища *(string)* – Название хранилища для загрузки и записи данных

```
class Test extends Contract {
  init(){
    super.init();
    //Создаём/загружаем хранилище
    this._db = new BlockchainArraySafe('name');
  }

  doSomething(){
    this._db.push('Hello');
  }

  doSomethingNext(){
   this._db.push('world!');
  }

  test(){
    this.doSomething();
    this.doSomethingNext();
    console.log(this._db.join(' ')); //Выведет "Hello world!"
  }
}
```

4.1.2.0. BlockchainMap

Объектно-подобная структура. Не сериализуема. Сохраняет типы объектов при условии, что их можно сериализовать.

new BlockchainMap (название хранилища)
Аргументы:

- название хранилища (string) – Название хранилища для загрузки и записи данных

```
class Test extends Contract {
  init(){
    super.init();
    //Создаём/загружаем хранилище
    this._db = new BlockchainMap('name');
    this._db['hello'] = 'Hello';
    this._db['world'] = 'world!';
    console.log(this._db['hello']+' '+this._db['world']); //Выведет "Hello world!"
  }

}
```

4.1.1.6. TokenRegister

Класс, реализующий безопасную логику работы реестра токенов. Является основой для TokenContract (п.4.1.3.2) и большинства токенов в системе. Содержит основные методы по управлению балансами кошельков, переводами, поддержкой минтинга, сжигания токенов, и учета существующего объема. Работает с числами до 256 знаков.

new TokenRegister *(название хранилища)*
Аргументы:

- название хранилища *(string)* – Название хранилища для загрузки и записи данных

balanceOf *(адрес)*
Аргументы:

- адрес (string) – Адрес для проверки баланса

Returns BigNumber:
Баланс кошелька

totalSupply *()*
Returns BigNumber:
Суммарный объем существующих токенов

setBalance *(адрес, баланс)*
Аргументы:

- адрес – Адрес для установки баланса
- баланс – Баланс, который необходимо установить

withdraw (адрес, сумма)
Аргументы:

- адрес – Адрес для списания токенов
- сумма – Сумма списания

minus *(адрес, сумма)*
Аргументы:

- адрес – Адрес для списания токенов
- сумма – Сумма списания

deposit *(адрес, сумма)*
Аргументы:

- адрес – Адрес для начисления токенов
- сумма – Сумма начисления

plus *(адрес, сумма)*
Аргументы:

- адрес – Адрес для начисления токенов
- сумма – Сумма начисления

transfer *(адрес отправителя, адрес получателя, сумма)*
Аргументы:

- отправителя (адрес) – Адрес списания токенов
- получателя (адрес) – Адрес начисления токенов

- сумма – Сумма перевода

burn *(адрес, сумма)*
Аргументы:
- адрес – Адрес для сжигания токенов
- сумма – Сумма сжигания

mint *(адрес, сумма)*
Аргументы:
- адрес – Адрес для выпуска токенов
- сумма – Сумма выпуска

```
class Test extends Contract {
  init(){
    super.init();
    this._wallets = new TokensRegister('TKN');
    this._wallets.mint('ME',1000);
    console.log(this._wallets.balanceOf('ME').toFixed()); //1000

    this._wallets.transfer('ME','FRIEND',100);
    console.log(this._wallets.balanceOf('ME').toFixed()); //900
    console.log(this._wallets.balanceOf('FRIEND').toFixed()); //100
  }

}
```

4.1.2. События

Генерируемые события используются для построения индекса событий контракта. Например, база данных событий позволяет быстро узнать баланс токенов на определенном кошельке на момент появления определенного блока.

4.1.2.1. Events

Events - глобальный объект, реализующий возможность создания (выпуск) события смарт-контракта. Метод emit испускает событие. Внимание: Мы не рекомендуем использовать объект Events напрямую. Используйте объект Event для создания экземпляра события и его выпуска.

emit *(событие, аргументы)*
Аргументы:
- событие *(string)* – Название события
- аргументы *(array)* – До 10 аргументов события

4.1.2.2. Event

Класс-обертка, реализующий безопасное и удобное использование объекта Events (п.4.1.2.1). Event реализует проверку типов входящих параметров, а также корректно приводит данные к верному формату.

new Event *(событие, массив типов аргументов)*
Аргументы:
- событие *(string)* – Название события

- массив типов аргументов *(array)* – До 10 типов аргументов события („number“,“string“,“array“,“object“)

emit *(аргументы)*
Аргументы:
- аргументы *(array)* – До 10 аргументов события, соответствующих ранее указанному типу

```
class Test extends Contract {
  init(){
    super.init();
    this._helloEvent = new Event('Hello',['string']);
    this._helloEvent.emit('Hello world!');
  }

}
```

Event может принимать не более 10 параметров в качестве входных аргументов.

4.1.3. Объекты контрактов

4.1.3.1. Contract

Объект, рекомендуемый к наследованию всеми контрактами платформы. Объект Contract реализует:

1. Базовый функционал проверки доступа.
2. Определение состояния запуска (deploy).
3. Прием и подтверждение оплаты.
4. Взаимодействие с системой межконтрактных взаиморасчетов
5. Интерфейс внешнего приложения

Внутренний метод init инициализирует внутренний функционал контракта, в том числе определение состояния запуска (deploy).
super.init()

Вызывается единожды в жизни смарт контракта, позволяет определить момент запуска (deploy)
deploy()

Позволяет получить информацию об оплате (см. Pay Object п.7.2):
payProcess()
Returns object: Объект с информацией оплаты

Проверка на то, что вызывающий является владельцем контракта:
assertOwnership([*msg*])
Аргументы:
- msg (*string*) – Сообщение на случай ошибки проверки

Проверка на то, что вызывающий является мастер-контрактом:
assertMaster([*msg*])
Аргументы:
- msg (*string*) – Сообщение на случай ошибки проверки

Проверка на то, что метод вызван извне (т.е. не из другого контракта):
assertExternal([*msg*])
Аргументы:

- msg (*string*) – Сообщение на случай ошибки проверки

Проверка на то, что метод вызван с передачей информации об оплате:
assertPayment([*msg*])
Аргументы:

- msg (*string*) – Сообщение на случай ошибки проверки

Добавление инфо-метода для внешнего приложения:
_registerAppInfoMethod(*имя метода, типы аргументов*)
Аргументы:

- имя метода (string) – Название метода
- типы аргументов (array) – Типы принимаемых значений метода

Добавление исполнительного-метода для внешнего приложения:
_registerAppDeployMethod(*имя метода, типы аргументов*)
Аргументы:

- имя метода (string) – Название метода
- типы аргументов (array) – Типы принимаемых значений метода

Добавление исходного кода приложения:
_registerApp(*исходный код*[, тип = "*web*"])
Аргументы:

- исходный код (*string*) – Исходный код внешнего приложения или информация для его поиска
- тип (*array*) – Тип внешнего приложения

Получение данных внешнего приложения:
getAppData()
Returns string: Сериализованный объект внешнего приложения

Регистрация обратного вызова для межконтрактных-покупок:
_registerC2CResultCallback(*адрес контракта продавца, функция обратного вызова*)
Аргументы:

- контракта продавца (*адрес*) – Адрес контракта-продавца, у которого сделан заказ
- обратного вызова (*функция*) – Функция, которая будет вызвана когда придёт заказ function(объект результата, номер заказа, адрес продавца)

4.1.3.2. TokenContract

Объект, рекомендуемый к наследованию всеми контрактами-токенами. Реализует интерфейс контракта стандарта IZ3 Token (см.п.7.1).

Свойство _wallets - экземпляр объекта TokenRegister (п.4.1.1.6).

Свойство _TransferEvent - экземпляр объекта Event (п.4.1.2.2).

Свойство _MintEvent - экземпляр объекта Event.

Свойство _BurnEvent - экземпляр объекта Event.

super.init(*initialEmission*[*, mintable = false*])

Аргументы:

- initialEmission – Стартовая эмиссия токенов
- mintable (boolean) – Разрешен ли минтинг в будущем

_getSender()
Результат: Адрес отправителя вызова

balanceOf(*address)*
Аргументы:

- address (string) – Адрес владельца токенов

Returns string:
Баланс токенов

totalSupply()
Returns string:
Количество всего выпущенных токенов

transfer(*to, amount*)
Аргументы:

- to (string) – Получатель токенов
- amount (string) – Количество токенов

burn(*amount*)
Аргументы:

- amount (string) – Количество сжигаемых токенов

mint(*amount*)
Аргументы:

- amount (string) – Количество выпускаемых токенов

getActionFee(*action, args*)
Аргументы:

- action (string) – Действие
- args (array) – Аргументы

Returns string:
Стоимость действия

getTransferFee(*amount*)
Аргументы:

- amount (string) – Сумма перевода

Returns string:
Стоимость перевода

4.1.4. Коннекторы

Коннекторы EcmaContracts - это классы-обёртки, реализующие упрощенный интерфейс взаимодействия с внешними контрактами определённых типов.

4.1.4.1. ContractConnector

Базовый класс для коннекторов. Позволяет регистрировать методы и их алиасы.

4.1.4.2. TokenContractConnector

Класс коннектора, реализующий взаимодействие с контрактом-токеном из другого контракта

new TokenContractConnector(*address*)
Аргументы:

- address (string) – Адрес контракта, к которому происходит подключение

balanceOf(*address*)
Аргументы:

- address (string) – Адрес владельца токенов

Returns string:
Баланс токенов

totalSupply()
Returns string: Количество всего выпущенных токенов

transfer(*to, amount)*
Аргументы:

- to (string) – Получатель токенов
- amount (string) – Количество токенов

burn(*amount*)
Аргументы:

- amount (string) – Количество сжигаемых токенов

mint(*amount*)
Аргументы:

- amount (string) – Количество выпускаемых токенов

```
class Test extends Contract {
  init(){
    super.init();
    let tokenContract = new TokenContractConnector(1);
    console.log(tokenContract.balanceOf('INVALID_ADDR')); //Выведет 0
  }

}
```

4.1.4.3. SellerContractConnector

SellerContractConnector - это класс коннектора, реализующий взаимодействие с контрактом-продавцом. Контракты-продавцы - это контракты, продающие данные или услугу внутри сети.

getPrice(*args*)
Аргументы:

- args (*array*) – Аргументы заказа

Результат:

Цена заказа

buy(*args*)
Аргументы:
- args (*array*) – Аргументы заказа

Результат:
id заказа

getResult(*orderId*)
Аргументы:
- orderId (*string*) – id заказа

Результат:
Объект содержащий заказ

4.1.4.4. Require

Require - не относится к стандартным коннекторам, однако реализует сходный функционал. Require позволяет создать экземпляр класса внешнего контракта с произвольными методами и свойствами. Наличие методов и свойств во внешнем классе проверяется во время выполнения кода.

new Require(*externalContract*)
Аргументы:
- externalContract (*string*) – Адрес подключаемого контракта

```
class Test extends Contract {
  init(){
    super.init();
    let tokenContract = new Require(1);
    console.log(tokenContract.balanceOf('INVALID_ADDR')); //Выведет 0
    console.log(tokenContract.invalidMethod('INVALID_ADDR')); //Бросит исключение
  }

}
```

Альтернативный синтаксис с методом-фабрикой:

require(*externalContract)*
Аргументы:
- externalContract (*string*) – Адрес подключаемого контракта

4.1.5. Системные классы

4.1.5.1. assert

Предоставляет функционал проверки каких - либо условий. Невыполнение этих условий ведет к выбросу исключения, и завершению вызова контракта с ошибкой. Состояние контракта откатывается до состояния до вызова.

Предоставляются следующие методы для проверки условий:

1. Простая проверка на истину

assert(*условие*[, *сообщение*])

Аргументы:

- условие – условие, истина
- сообщение (*string*) – сообщение

2. Проверка на несоответствие типу undefined

defined(*условие*[, *сообщение*])

Аргументы:

- условие – простое условие с использованием условных операторов
- сообщение (*string*) – сообщение при неверности условия

3. Проверяет верность условия a > b

gt(*a, b*[, *сообщение*])

Аргументы:

- a – числовой параметр
- b – числовой параметр
- сообщение (*string*) – сообщение при неверности условия

4. Проверяет верность условия a < b

lt(*a, b*[, *сообщение*])

Аргументы:

- a – числовой параметр
- b – числовой параметр
- сообщение (*string*) – сообщение при неверности условия

5. Проверяет истинность условия. Алиас assert.assert

true(*условие*[, *сообщение*])

Аргументы:

- условие – условие
- сообщение (*string*) – сообщение при неверности условия

6. Проверка ложности условия (аналогично «Неверно, что…»)

false(*условие*[, *сообщение*])

Аргументы:

- условие – условие; условие
- сообщение (string) – сообщение при неверности условия

```
class Test extends Contract {
   init(){
      super.init();
      assert.gt(100,200,'100 не может быть больше 200');
   }

}
```

4.1.5.2. contracts

Предоставляет функционал для управления текущим или взаимодействия с внешними контрактами. Доступны следующие методы:

Вызов метода другого контракта с сохранением измененного состояния.

callMethodDeploy(*адресКонтракта*, *метод*[, *аргументы*])
Аргументы:

- адресКонтракта (*string*) – адрес вызываемого контракта.
- метод (*string*) – метод вызываемого контракта.
- аргументы (*array*) – передаваемый методу аргументы в виде массива.

Бросает исключение:

- Error1 – „You can't call method from himself" - вызов изнутри.
- Error2 – „External call failed" - ошибка при вызове метода.

Результат:

- результат вызова метода.

Вызов метода другого контракта выполняемый после завершения текущего вызова:

callDelayedMethodDeploy(*адресКонтракта*, *метод*[, *аргументы*])
Аргументы:

- адресКонтракта (string) – адрес вызываемого контракта.
- метод (string) – метод вызываемого контракта.
- аргументы (array) – передаваемый методу аргументы в виде массива.

Бросает исключение:

- Error1 – „You can't call method from himself" - вызов изнутри.

Запрос свойства другого контракта:

getContractProperty(*адресКонтракта*, *свойство*)
Аргументы:

- адресКонтракта (string) – адрес вызываемого контракта.
- свойство (string) – свойство контракта

Бросает исключение:

- Error1 – „You can't call method from himself" - вызов изнутри.

Результат:

- Значение свойства

getMasterContractAddress()
Результат:

- Адрес мастер-контракта, определенный в конфигурации

caller()
Результат:

- адрес вызывающего контракта; false, если вызов внешний

callingIndex()
Результат:

- номер текущего вызова контракта в цепочке вызовов или 0, если вызов внешний

isChild()
Результат::

- true, если контракт был вызван другим контрактом, в ином случае - false

isDeploy()
Результат:
- true, если контракт находится в состоянии запуска (deploy), в ином случае - false

isDelayedCall()
Результат:
- true если вызов выполнен с помощью callDelayedMethodDeploy

getExtendedState()
Результат:
- Объект добавочного состояния

4.1.5.3. crypto

Методы работы с криптографией

hash(*data*)
Аргументы:
- data (string) – Данные для хеширования

Результат:
- Строковое представление хеш-суммы

Проверка цифровой подписи сконфигурированной функцией:

verifySign(*data*, *sign*, *publicKey*)
Аргументы:
- data (string) – Данные для проверки
- sign (string) – Подпись данных
- publicKey (string) – Публичный ключ

Returns boolean:
- true если проверка успешна

4.1.5.4. global

Другие глобальные методы.

Передача класса контракта в виртуальную машину:

registerContract(*контракт*)
Аргументы:
- data (string) – Данные для проверки
- sign (string) – Подпись данных
- publicKey (string) – Публичный ключ

Returns boolean:
- true если проверка успешна

Вывод данных на экран (при условии включенного вывода)

console.log(*данные*)
Аргументы:
- данные (*string*) – Данные для проверки

getState()
Returns object:

- Текущий объект state. Рекомендуется использовать вместо global.state

4.2. Другие доступные объекты

4.2.1. BigNumber

Объект, позволяющий безопасно проводить арифметические операции с большими числами.
Рекомендуется для любых вычислений связанных с балансами пользователей, и с необходимой точностью.

Описание BigNumber доступно по ссылке: https://github.com/MikeMcl/bignumber.js/

4.2.2. Date

Встроенный объект даты, однако ход времени объекта Date заморожен, и установлен в соответствии с состоянием, передаваемым в вызове.

4.2.3. Math

Встроенный объект. Отличием от стандартного Math является реализация random: использует внешний seed, устанавливаемый в соответствии с состоянием, передаваемым в вызове.

4.3. API

Для взаимодействия с контрактом с помощью API используются методы:

- **POST** contracts/ecma/deployContract
- **GET** contracts/ecma/getContractInfo/*:contractAddress:*
- **GET** contracts/ecma/getContractProperty/*:contractAddress:*/*:property:*
- **POST** contracts/ecma/callMethod/*:contractAddress:*/*:method:*
- **POST** contracts/ecma/deployMethod/*:contractAddress:*/*:method:*
- **POST** contracts/ecma/deploySignedMethod/*:contractAddress:*

POST contracts/ecma/deployContract
Запуск нового контракта в сеть
POST параметры:

- **resourceRent** - Число (от 0 и более). Количество токенов выделяемых для аренды ресурсов
- **source** - Строка. Исходный JavaScript код смарт контракта

GET contracts/ecma/getContractInfo/:contractAddress
Получение информации о контракте
GET параметры:

- **contractAddress** - Строка. Адрес контракта

GET contracts/ecma/getContractProperty/:contractAddress:/:property
Получение значения свойства контракта
GET параметры:

- **contractAddress** - Строка. Адрес контракта
- **property** - Строка. Название свойства

POST contracts/ecma/callMethod/:contractAddress:/:method:
Вызов метода контракта, с последующим откатом изменений, и возвратом результата вызова или ошибки
GET параметры:

- **contractAddress** - Строка. Адрес контракта
- **method** - Строка. Название вызываемого метода

POST параметры:

- **argsEncoded** - Строка. JSON сериализованное значение со списком аргумента вызова

POST contracts/ecma/deployMethod/:contractAddress:/:method:
Вызов метода контракта, с последующей записью изменений в цепочку блоков. Возвращает содержимое нового блока или ошибку.
GET параметры:

- **contractAddress** - Строка. Адрес контракта
- **method** - Строка. Название вызываемого метода

POST параметры:

- **argsEncoded** - Строка. JSON сериализованное значение со списком аргумента вызова

POST contracts/ecma/deploySignedMethod/:contractAddress:
Вызов метода контракта, с последующей записью изменений в цепочку блоков. В отличии от deployMethod принимает на вход подписанный блок EcmaContractCallBlock
GET параметры:

- **contractAddress** - Строка. Адрес контракта

POST параметры:

- **source** - Строка. JSON сериализованный объект подписанного блока

6. DApp

Децентрализованные приложения в IZZZIO позволяют создавать добавочный функционал для ноды. Такие приложения имеют широкий функционал:

1. Взаимодействие с цепочкой блоков, предобработка, реакция на изменения цепочки
2. Взаимодействие со смарт контрактами, в том числе запуск новых
3. Взаимодействие с другими децентрализованными приложениями посредством встроенных протоколов. См. StarWave (п.6.7), StarWaveCrypto (п.6.8)
4. Интеграция централизованных приложений с децентрализованным.

Для написания децентрализованных приложений в IZZZIO используется класс DApp. Класс предоставляет все необходимые методы для работы с функционалом сети, и обеспечивает совместимость приложений с разными версиями узлов блокчейна IZZZIO.

6.1. Класс DApp

При создании класса наследуемого от DApp рекомендуется не объявлять конструктор, и использовать вместо него встроенный метод init Для обработки состояния завершения используется метод terminate(cb)

6.2. Глобальные переменные

Для использования в DApp доступны следующие глобальные переменные:

1. global.PATH.mainDir - путь до корневой директории файлов блокчейна IZZZIO
2. global.PATH.configDir - путь до директории конфигурационного файла

Эти переменные обычно используются для включения файлов модулей ядра в проект.

const DApp = require(global.PATH.mainDir + '/app/DApp'); //Поиск DApp будет выполнен в корневой директории блокчейна

```
const DApp = require(global.PATH.mainDir + '/app/DApp'); //Поиск DApp будет выполнен в корневой директории блокчейна

class TestDapp extends DApp {
   init(){
      console.log('DApp ready!');
   }

   terminate(cb){
      console.log('DApp terminated');
   }

}
```

6.3. Доступные объекты

6.3.1. logger

Внутренний путь: modules/logger

logger обеспечивает стандартизацию вывода сообщений в консоль.

Свойство: disable - true - отключение вывода сообщений; false - вывод включения

new Logger([*prefix*])
Аргументы:

- prefix (*string*) – Префикс, с которым будет выводится сообщение

getPrefix()
Результат:

Текущий установленный префикс

log(*type, data*)
Аргументы:

- type (*string*) – Тип сообщения для вывода
- data (*string*) – Сообщение

info(*type, data)*
Аргументы:

- data (*string*) – Сообщение информационного типа

init(*type, data*)
Аргументы:

- data (*string*) – Сообщение инициализации

error(*type, data*)
Аргументы:

- data (*string*) – Сообщение об ошибке

fatal(*type, data)*
Аргументы:

- data (*string*) – Сообщение о критической ошибке

warning(*type, data*)
Аргументы:

- data (*string*) – Предупреждение

```
const DApp = require(global.PATH.mainDir + '/app/DApp');
const logger = new (require(global.PATH.mainDir + '/modules/logger'))("TestDApp");

class TestDApp extends DApp {
   init(){
      logger.info('Test DApp ready!'); //Выведет "Fri, 01 Feb 2019 14:12:48 GMT Info:  ECMAContract: Test DApp ready"
   }

}
```

6.3.2. assert

Внутренний путь: modules/testing/assert

Предоставляет функционал проверки каких - либо условий. Невыполнение этих условий ведет к выбросу исключения, и завершению вызова контракта с ошибкой. Состояние контракта откатывается до состояния до вызова.

Реализация идентична реализации для EcmaContract (п.4).

Предоставляются следующие методы для проверки условий:

1. Простая проверка на истину

assert(*условие*[, *сообщение*])
Аргументы:

- условие – условие, истина
- сообщение (*string*) – сообщение

2. Проверка на несоответствие типу undefined

defined(*условие*[, *сообщение*])
Аргументы:

- условие – простое условие с использованием условных операторов
- сообщение (*string*) – сообщение при неверности условия

3. Проверяет верность условия a > b

gt(*a, b*[, *сообщение*])

Аргументы:

- a – числовой параметр
- b – числовой параметр
- сообщение (*string*) – сообщение при неверности условия

4. Проверяет верность условия a < b

lt(*a, b[, сообщение*])

Аргументы:

- a – числовой параметр
- b – числовой параметр
- сообщение (string) – сообщение при неверности условия

5. Проверяет истинность условия. Алиас assert.assert

true(*условие*[, *сообщение*])

Аргументы:

- условие – условие
- сообщение (string) – сообщение при неверности условия

6. Проверка ложности условия (аналогично «Неверно, что…»)

false(*условие*[, *сообщение*])

Аргументы:

- условие – условие; условие
- сообщение (*string*) – сообщение при неверности условия

```
class Test extends Contract {
   init(){
      super.init();
      assert.gt(100,200,'100 не может быть больше 200');
   }

}
```

6.3.3. instanceStorage

Внутренний путь: modules/instanceStorage

Предоставляет доступ к созданным экземплярам объектов на основе ключей.

instanceStorage.get(*objName*)

Аргументы:

- name (string) – Название сохраненной сущности

Результат:

Сохраненная сущность или null при отсутствии

instanceStorage.put(*objName, value)*

Аргументы:

- name (*string*) – Название сохраняемой сущности
- value – Объект сущности

По умолчанию instanceStorage содержит следующие сущности:

1. Blockchain - главный экземпляр объекта сети
2. config - экземпляр объекта конфигурации
3. wallet - текущий кошелек
4. blocks - хранилище данных блоков
5. blockHandler - экземпляр объекта обработчика блоков
6. frontend - экземпляр объекта RPC интерфейса и оболочки
7. transactor - экземпляр объекта контроля транзакций
8. cryptography - экземпляр объекта криптографии
9. blockchainObject - алиас объекта Blockchain
10. accountManager - менеджер аккаунтов

Опциональные объекты и значения:

1. ecmaContract - экземпляр объекта смарт контрактов (если включены)
2. dapp - экземпляр объекта децентрализованного приложения (если включено)
3. terminating - флаг завершения работы

```
const DApp = require(global.PATH.mainDir + '/app/DApp');
const storj = require(global.PATH.mainDir + '/modules/instanceStorage');

class TestDApp extends DApp {
   init(){
      console.log(storj.get('config').p2pPort); //6015
   }

}
```

6.3.4. StarWaveCrypto

Внутренний путь: modules/starwaveCrypto

StarWaveCrypto - модуль, реализующий создание шифрованного канала для обмена данными между узлами сети. При создании подключения, между узлами происходит генерация и обмен ключами по протоколу Диффи - Хеллмана (DH).

Для создания шифрованного канала используется метод, осуществляющий «рукопожатие». Таким образом, принимающая сторона, используя метод обработки получаемых сообщений, сразу создаст новое защищенное соединение, или использует существующее. Пример установки соединения:

```
const DApp = require(global.PATH.mainDir + '/app/DApp');
const StarwaveCrypto = require(global.PATH.mainDir + '/modules/starwaveCrypto');

class TestDApp extends DApp {

   init() {

      let that = this;
```

```
    starwave.registerMessageHandler('SW_TEST', function (message) {
      console.log('New message', message);
    });

    let crypto =  new StarwaveCrypto(this.starwave, this.blockchain.secretKeys).makeConnection(RECEIVER_ADDRESS, function(who,
secretKey){
      console.log('Connected!');
      let message = that.starwave.createMessage('Hello Node Two' + Math.random(), 'RECEIVER_ADDRESS', undefined, 'SW_TEST');
      crypto.sendMessage(m);
    });
  };
}
```

6.4. Коннекторы DAPP

Коннекторы DApp - классы-обёртки, реализующие упрощенный интерфейс взаимодействия с контрактами определённых типов.

6.4.1. DApp-ContractConnector

Внутренний путь: modules/smartContracts/connectors/ContractConnector

Базовый абстрактный класс для коннекторов. Позволяет регистрировать методы и их алиасы.

6.4.2. DApp-TokenContractConnector

Внутренний путь: modules/smartContracts/connectors/TokenContractConnector

DApp-TokenContractConnector - это класс коннектора, реализующий взаимодействие с контрактом-токеном для DApp

new TokenContractConnector(*EcmaContractsInstance, contractAddress*)
Аргументы:

- EcmaContractsInstance – Экземпляр объекта EcmaContracts
- contractAddress (*string*) – Адрес контракта, к которому происходит подключение

async balanceOf(*address*)
Аргументы:

- address (string) – Адрес владельца токенов

Returns string:

Баланс токенов

async totalSupply()
Returns string:

- Количество всего выпущенных токенов

async transfer(*to, amount*)
Аргументы:

- to (string) – Получатель токенов
- amount (string) – Количество токенов

async burn(*amount*)
Аргументы:

- amount (string) – Количество сжигаемых токенов

async mint(*amount*)
Аргументы:

- amount (string) – Количество выпускаемых токенов

async contract()
Returns string:

- Свойство contract из контракта. Данные о контракте

Выполнение метода из другого контракта с оплатой:
async pay(*address, method, txAmount, args*)
Аргументы:

- address (string) – Адрес контракта для вызова метода
- method (string) – Метод
- txAmount (string) – Сумма перевода
- args (array) – Аргументы вызова

Результат:
Новый созданный блок

```
const DApp = require(global.PATH.mainDir + '/app/DApp');
const TokenContractConnector = require(global.PATH.mainDir + '/modules/smartContracts/connectors/TokenContractConnector');

class TestDApp extends DApp {
   async init(){
      let mainToken = new TokenContractConnector(this.ecmaContract, this.getMasterContractAddress());

      console.log(await mainToken.balanceOf(this.getCurrentWallet().id)); //Выведет баланс текущего кошелька

      //Вызовет метод processPayment из контракта SOME_CONTRACT_ADDRESS с параметром 'Hello', передав состояние проведения оплаты на 1 токен с текущего кошелька
      await mainToken.pay(SOME_CONTRACT_ADDRESS, "processPayment", '1', ['Hello']);
   }

}
```

6.5. Внутренние объекты DApp

6.5.1. DApp.network

Предоставляет базовый функционал взаимодействия с сетевым интерфейсом, а также регистрацию RPC и обработчиков интерфейса.

1. Возвращает массив текущих пиров в формате ws://address:port

getCurrentPeers()
Результат:
Список подключенных пиров

2. Получение сокета по адресу шины (адрес внешней ноды)

getSocketByBusAdress(*address*)
Аргументы:

- address (string) – адрес шины

Результат:

объект сокета или false, если адрес не был найден.

3. Прямая отправка сообщения в сокет

socketSend(*ws, message)*
Аргументы:

- ws – сокет.
- message (*string*) – сообщение.

4. Регистрация метода обратного вызова для get запроса

rpc.registerGetHandler(*url*, *callback*)
Аргументы:

- url (*string*) – Обрабочтик ссылки
- callback (*function*) – Метод обратного вызова

5. Регистрация метода обратного вызова для post запроса

rpc.registerPostHandler(*url*, *callback*)
Аргументы:

- url (*string*) – Обрабочтик ссылки
- callback (*function*) – Метод обратного вызова

Методы rpc.registerGetHandler и rpc.registerPostHandler основаны на использовании фреймворка Express.js (Подробнее об обработчиках в документации Express.js: https://expressjs.com/ru/4x/api.html)

6.5.2. DApp.messaging

Устаревший функционал. Предоставляет базовый функционал обмена сообщениями между узлами сети.

1. Регистрация обработчика сообщений

registerMessageHandler(*message*, *handler*)
Аргументы:

- message (*string*) – идентификатор сообщения
- handler (*function*) – функция - обработчик сообщений

2. Метод широковещательной передачи сообщений по сети

broadcastMessage(*data*, *message*, *receiver*)
Аргументы:

- data – передаваемый объект
- message (*string*) – идентификатор сообщения
- receiver (*string*) – получатель сообщения

3. Метод передачи сообщения ближайшему получателю

sendMessage(*data*, *message*, *receiver*)
Аргументы:

- data – объект, содержащий само сообщение.
- message (*string*) – идентификатор сообщения.
- receiver (*string*) – получатель сообщения.

Методы DApp.messaging.starwave описаны в разделе StarWave (п.6.7)

6.5.3. DApp.blocks

DApp.blocks - предоставляет доступ к функционалу генерации и добавления блоков, обработчику событий.

Генерирует блок без добавления в цепочку блоков:
generateBlock(*blockData*, *cb*, *cancelCondition*)
Аргументы:

- blockData (*Block*) – Подписанный блок Block
- cb (*function*) – функция обратного вызова function(newBlock)
- cancelCondition – Функция условия отмены генерации блока

Генерация блока с добавлением в цепочку блоков и уведомлением узлов о новом блоке:
generateAndAddBlock(*blockData*, *cb*, *cancelCondition)*
Аргументы:

- blockData (Block) – Подписанный блок Block
- cb (function) – функция обратного вызова function(newBlock)
- cancelCondition – Функция условия отмены генерации блока

Добавление сгенерированного блока:
addBlock(*newBlock*, *cb*)
Аргументы:
newBlock – Готовый блок
cb – callback - обратного вызова

6.5.3.1. Методы обработчика блоков

Возвращает экземпляр текущего обработчика блоков:
handler.get()
Результат:
объект обработчика (ловца) блоков.

Регистрация обработчика блока по типу:
handler.registerHandler(*type*, *handler*)
Аргументы:

- type (*string*) – Тип блока
- handler (*function*) – функция - обработчик блоков function(blockData, blockSource, callback)

Вызов callback в обработчике блоков обязателен. **Отсутствие обратного вызова приведёт к зависанию обработчика блоков.**

6.5.4. DApp.contracts

DApp.contracts предоставляет функционал доступа к модулю EcmaContracts.

contracts.ecma - предоставляет прямой доступ к объекту EcmaContracts. См. EcmaContracts

contracts.ecmaPromise - предоставляет основные методы EcmaContracts для использования с асинхронным режимом (async-await)

Выполняет запуск контракта в сети:

ecmaPromise.deployContract(*source*, *resourceRent*)
Аргументы:

- source (*string*) – Исходный код контракта
- resourceRent (*number*) – Количество токенов, выделяемых на аренду ресурсов контракта

Результат:

объект нового блока, содержащего контракт

Выполняет запуск метода из контракта:
ecmaPromise.deployMethod(*address*, *method*, *args*, *state*)
Аргументы:

- address (*string*) – Адрес контракта
- method (*string*) – Метод контракта
- args (*array*) – Массив с аргументами вызова
- state (*object*) – Объект передаваемого состояния

Результат:

объект нового блока

Выполняет запуск метода из контракта с откатом состояния
ecmaPromise.callMethodRollback(*address*, *method*, *args*, *state*)
Аргументы:

- address (*string*) – Адрес контракта
- method (*string*) – Метод контракта
- args (*array*) – Массив с аргументами вызова
- state (*object*) – Объект передаваемого состояния

Результат:

Результат выполнения метода контракта

6.6. Внутренние методы DApp

DApp.getConfig()
Результат:

объект конфигурации

DApp.getBlockHandler()
Результат:

объект обработчика блоков

DApp.getCurrentWallet()
Результат:

объект текущего кошелька

DApp.getMasterContractAddress()
Результат:

адрес мастер-контракта

6.7. StarWave

StarWave - это протокол высокоскоростной передачи данных внутри сети IZZZIO. Протокол автоматически составляет кратчайший маршрут доставки сообщения до получателя. В случае нарушения маршрута, происходит его перестроение по такому-же принципу.

Методы StarWave доступны в DApp DApp.messaging.starwave

Методы messaging.starwave

Регистрация обработчика сообщений:
starwave.registerMessageHandler(*message*, *handler*)
Аргументы:

- message (*string*) – идентификатор сообщения
- handler (*function*) – функция - обработчик сообщений

Метод передачи сообщения ближайшему получателю
starwave.sendMessage(*data*, *message*, *receiver*)
Аргументы:

- data – объект, содержащий само сообщение.
- message (*string*) – идентификатор сообщения.
- receiver (*string*) – получатель сообщения.

Метод создания структуры сообщения
starwave.createMessage(*data*, *receiver*, *sender = undefined*, *id*, *timestamp = undefined*, *TTL = undefined*, *relevancyTime = undefined*, *route = undefined*, *type = undefined*, *timestampOfStart*)
Аргументы:

- data – объект, содержащий само сообщение.
- receiver (*string*) – получатель сообщения
- sender (*string*) – отправитель сообщения; undefined - используются системное инфо об отправителе
- id (*string*) – Идентификатор сообщения
- timestamp (*number*) – Метка времени сообщения; undefined - автоматически
- TTL (*number*) – Возможное количество скачков сообщения; undefined - автоматически
- relevancyTime (*number*) – время актуальности сообщения; undefined - автоматически
- timestamp – Метка времени сообщения; undefined - автоматически
- route (*array*) – Маршрут сообщения; undefined - автоматически
- type (*string*) – Тип сообщения; undefined - автоматически
- timestampOfStart (*number*) – Метка времени отправки сообщения; undefined - автоматически

Результат:

Объект структуры сообщения

6.8. StarWaveCrypto

Внутренний путь: modules/starwaveCrypto

StarWaveCrypto - это модуль, реализующий создание шифрованного канала для обмена данными между узлами сети. При создании подключения, между узлами происходит генерация и обмен ключами по протоколу Диффи - Хеллмана (DH).

Для создания шифрованного канала используется метод, осуществляющий «рукопожатие». Таким образом, принимающая сторона, используя метод обработки получаемых сообщений, сразу создаст новое защищенное соединение, или использует существующее. Пример установки соединения:

7. Стандарты

7.1. IZ3 Token

Стандарт токена определяет обязательные условия:

1. Свойство contract возвращающее информацию о токене {name, ticker, owner, emission, type} и type="token"
2. Метод init([исходная эмиссия = 0 , разрешен минтинг = false])
3. Метод balanceOf(адрес)
4. Метод totalSupply() возвращающий общую сумму выпущенных токенов
5. Метод transfer(адрес получателя, сумма перевода)
6. Метод burn(сумма сжигания)
7. Метод mint(сумма минтинга)
8. Метод getTransferFee(сумма) передающий стоимость операции перевода
9. Метод getActionFee(действие, параметры) передающий стоимость выполнения какой-либо операции
10. События Transfer(адрес отправителя,адрес получателя, сумма) вызываемого при любом движении средств
11. События Mint(адрес, сумма)
12. События Burn(адрес, сумма)

Стандарт токена в полной мере реализован встроенным классом TokenContract (п.4.1.3.2).

7.2. Pay Object

Объект, возвращаемый методом payProcess класса Contract.

```
{
  amount,          //Сумма платежа, BigNumber
  rawAmount,       //Сумма платежа, исходный вид
  ticker,          //Тикер валюты платежа
  balance,         //Новый баланс контракта, BigNumber
  rawBalance,      //Новый баланс контракта, исходный вид
  caller,          //Адрес контракта токена платежа
  contractName,    //Имя контракта-токена платежа
}
```

Для проверки валюты платежа рекомендуется проверять адрес вызывающего таким образом:

```
assert.true(Number(PayObject.caller) === Number(WANTED_TOKEN_ADDR));
```

Для приема платежа только основного токена сети, используйте:

```
assert.true(Number(PayObject.caller) === Number(contracts.getMasterContractAddress()));
```

7.3. state

Объект текущего состояния запуска. Один из самых важных объектов виртуальной среды.

Рекомендованный метод получить текущий state
`const state = global.getState();`

Содержимое объекта может сильно меняться в зависимости от текущего состояния запуска контракта, поэтому рекомендуется проверять наличие или отсутствие необходимых свойств перед работой с ними.

Основные свойства:

1. state.from - Адрес пользователя, инициировавшего запуск методов смарт контракта. Видно всем контрактам в цепочке вызовов
2. state.block - Опциональное свойство. Объект блока текущего вызова (см. п. 7.4. Block)
   state.contractAddress - Адрес контракта текущего вызова.
3. state.extend - Опциональное свойство. Объект дополнительного состояния.
4. state.randomSeed - Опциональное свойство. Текущая настройка генератора случайных чисел
5. state.calledFrom - Опциональное свойство. Адрес контракта, совершившего текущий вызов
6. state.delayedMethod - Опциональное свойство. Флаг отложенного вызова
7. state.callingIndex - Опциональное свойство. Номер текущего вызова в цепочке вызовов

7.4. Block

Объект блока. Свойства:

1. index - текущий номер
2. previousHash - хеш предыдущего блока
3. timestamp - метка времени блока
4. startTimestamp - метка начала генерации блока
5. data - строка данных
6. hash - хеш блока
7. sign - цифровая подпись блока