FLEX Functional Specification

Prepared by Pruvendo 06/15/22

Executive summary

The purpose of the present document is to provide the detailed functional specification of the *FLEX* - decentralized stock exchange. The created specification is supposed to be turned into a formal specification and later used for the formal verification of the product.

This document considers the FLEX code from the repository https://github.com/tonlabs/flex/tree/main/flex published in commit record **ac95714f7126947bec12cbc948ac4e5751b73e4b**.

We studied the following smart-contracts *AuthIndex*, *Flex*, *FlexClient* , *PriceXchg* , *UserIdIndex*, *XchgPair*, as well as the TIP-3 ecosystem, all implemented in C++.

Github: https://github.com/tonlabs/flex
Core documentation: https://tonlabs.github.io/flex/[1]

Please address all the questions and comments to the Telegram account @SergeyEgorovSPb.

[1] The core documentation is very vague, has nothing but a few brief comments taken from code

Methodology

The paradigm of the present document follows the first steps of the Waterfall model used by Sun Microsystems (currently acquired by Oracle). These initial steps (specification part) contained three phases:

- Marketing part (Project Specification Document) that described the product from customer's point of view
- Architect part (Project Requirement Document) that described the product features to be used by an architect (being aware about smart-contract structure)
- Developer part (Functional Specification Document) that described an implementation-dependent low-level specification for the developers

While many verifiers think that the Developer part doesn't have a value, our team believes it's the essential part of the specification, as it allows us to easily construct formal statements to be verified. Otherwise, the link between the specification and formal statements will be lost.

Many verifiers dislike the specifications made from code, as, according to their opinion, they simply prove that the code corresponds to itself.

Indeed, in an ideal world this kind of specification MUST (as it was at Sun Microsystems mentioned above) be created before the code implementation, or, at least, some free-word specification must be created beforehand. Unfortunately, it's not the case for us: the specification provided is extremely vague, mostly based on brief comments from the source code, so the interpretation of the source code, rethinking it from the common sense perspective and reverse creation of the specification is the ONLY way we have.

Document structure

- Section I provides the business level (aka Project Concepts Document) of the ecosystem being described
- Section II provides the architect level (aka Project Specification Document) where there are properties from the customer's point of view are described and high-level architecture provided
- Section III provides the implementation-dependent properties (aka Functional Specification Document) where the contract structure is defined, as well as functionality of each contract
- Section IV describes the TIP-3 subsystem
- Section V describes the possible attacks (with their mitigation), as well as:
    - Business-level issues
    - Potential bugs found

Section I. Business-level description

I.1. System purpose

Flex is an Everscale-based decentralized trading system. It is implemented as a central limit order book (CLOB) over a distributed smart contract model, actually operated as a distributed limit order book (DLOB).

The system serves to trade with token pairs implemented using *TIP-3* technology (*TIP-3 - EVER* pairs are planned in future but currently not implemented). The system supports two basic trading operations: to buy and to sell tokens within the DLOB trading system, with an optional subscription to attach an automatic trade (beyond the current document). Both trading operations have varitiees to construct different market strategies.

I.2 Evaluation of market features (of the trading system and payment systems)[2]

The interaction at the market serves four key desirable features of a market:

- Price discovery
- Liquidity provision
- Reduction of search
- Information costs

Traders should be able to submit orders and observe in real-time an aggregated state of the market that supports price discovery. Market liquidity is supported by limit orders, which are rewarded from transaction fees of the exchange. Flex should provide complete information about updated prices and volumes, including market history. Information costs are low, due to usage of Everscale, where transactions are extremely cheap.

The current project must have an architecture of distributed systems. Key principles for their evaluation are[3]:

- scalability features,
- geographical allocation,
- administration.

Scalability has a few directions for the project: an increase in number of nodes, number of final clients, number of trading pairs. This feature is automatically achieved by the "no-dictionary"[4] paradigm that must be implemented.

[2] https://www.bundesbank.de/resource/blob/706630/85cb02d530f5dc9ec3d4382557bc694d/mL/2006-01-securities-market-data.pdf

[3] Neuman, B. Cli. "Scale in distributed systems." *ISI/USC* (1994): 68.

[4] A contract should not handle child entities in dictionaries but rather to deploy a separate contract for each child thus relying on its address being easily calculated rather than recording it in a dictionary.

Wide range of client geographical allocations is the intrincic property of blockchain systems so it’s achieved automatically.

Administration issues include decisions of DAO to introduce new trade pairs, speed of decisions to terminate trade, updating contracts’ code. The detailed analysis of such issues is provided at Section V.

I.3. Roles

There should be two roles in the project:

- Flex owner - the entity that created the Flex exchange (*Owner* or *DAO*)
- Flex client - any other user who can place an order (*Client*)

I.4. Limit orders

*A **limit order*** bounds execution price and is used for hedging trading positions. An order may be never executed and will expire after the provided time. Limit orders supply market liquidity by a permanent presence at the market, which reduces search costs of clients.

There are two types of limit orders.
Maker orders are just added to the order book. This kind of order improves liquidity of a market and an owner can be rewarded with a negative trading fee if it is completed. Taker orders are matched immediately against an existing order. They reduce market liquidity and are charged with positive trading fees.

Life-time of orders

An order may have a maturity to expire being unmatched. There are few options to be supported:
Good-til-canceled (GTC) order requires an explicit cancellation or expires within a predefined period. 90-day period is suggested.

Immediate or cancel (IOC) order is immediately executed or automatically canceled. The orders have partial execution, different from FOK orders.

I.5. Mechanics

An *Owner* can:

- Create a new Flex stock exchange
- Approve or reject request for new TIP-3 - TIP-3 trading pairs (thus registering them)

A *Client* can:

- Place an order
- Cancel an order
- Send a request to register a new TIP-3 - TIP-3 trading pair

The following rules must be in place:

- No slippage[5]
- FIFO - earlier orders must be resolved earlier
- If the client marks an order as to be immediately executed it must be immediately matched with counterparts and executed, and if it's fully or partially impossible, the "unexecuted part" must be returned and the order must be canceled

[5] Say, X wants to sell some equities for 9, at the same time Y wants to buy the same equities for 11. It sounds evident that they must meet each other and make a deal, say, for 10. This "meeting" is called "slippage". However, Flex is not designed to support it, so X and Y would not meet each other to make a deal. It's by design and not considered a bug.

- Otherwise, the “unexecuted part” must be put as a order into the order book[6] to wait for the counterpart request
- All the assets in the order book must be locked to prevent its usage before order execution or canceling

Section II. High-level properties

II.1. Terms

Documentation of the project uses several terms that are defined below.

Terms	**Definition**
Flex Exchange Root	A basic contract, which manages the trading pairs, collects fees from buyers and sellers for transactions.
FLEX AMM	A market maker, which holds positions in an order book. Development is pending.
Owner (or DAO)	The entity that created a new Flex contract
TIP-3 token	TIP-3 token, external to the system.
Liquidity Providers	Clients, which provide limit orders to the system and have a share in FLEX AMM profit.
TIP-3 wallet	An account in external token TIP-3.
Unit	A functional block of an architecture diagram. It unites operations with similar or closely tight goals.
Flex Root	The contract contains an exchange configuration

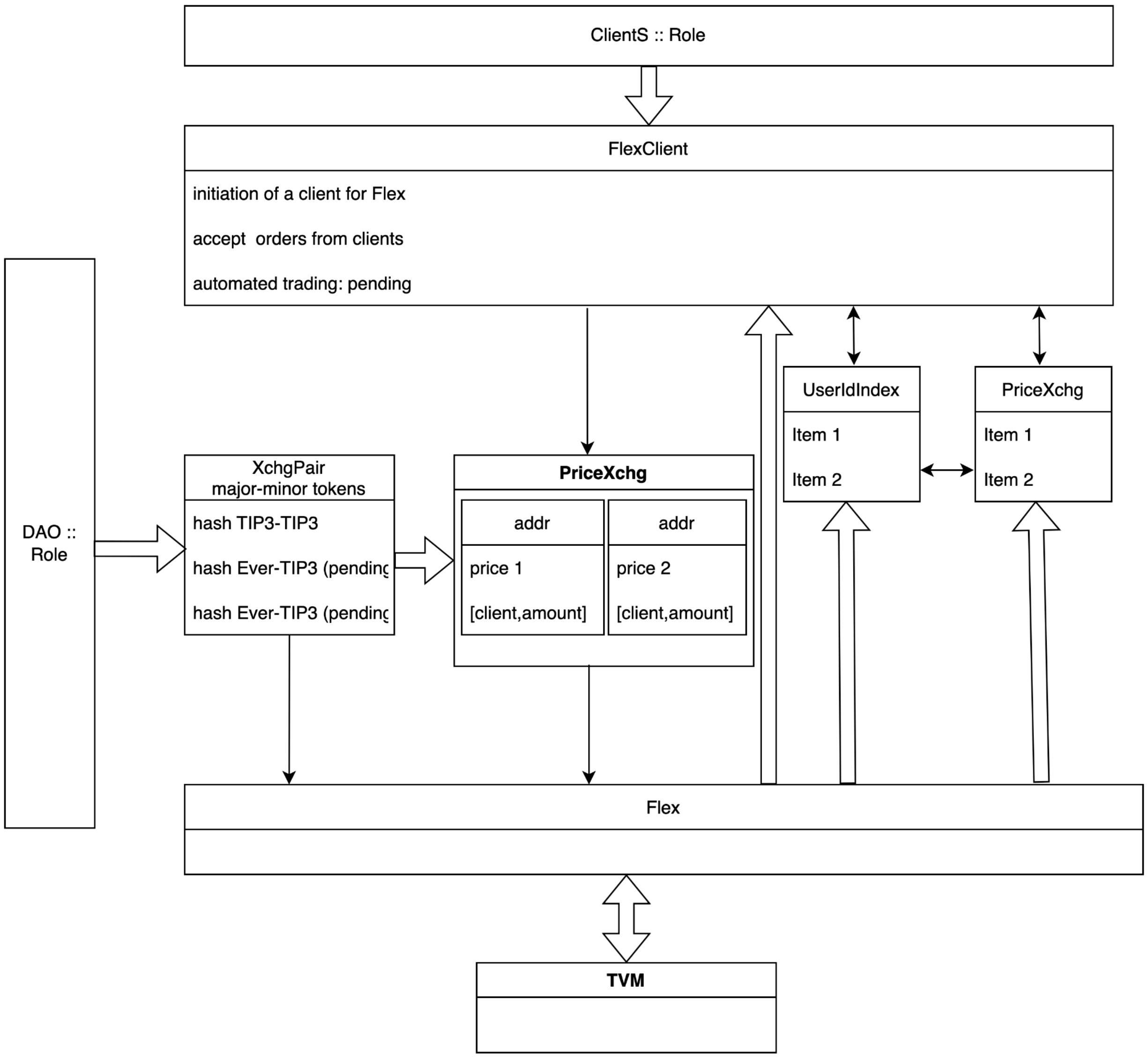
[6] Order book is an ordered (FIFO) set of requests to be executed

Wrapper (Flex Root token)	The contract converts external tokens into Flex TIP-3 tokens with all accompanying operations.
Exchange Pair (XchgPair)	The contract defines a trading pair of to=kens.
Price Exchange (PriceXchg)	The contract enables a trade for a pair of tokens at specific price.
Flex Client (FlexClient)	The contract interface between a user and the Flex system.
Flex TIP3 token wallet	The contract holds client tokens within Flex.
Order	A message from a client to sell or buy tokens, where a client indicates a price, an amount, and immediacy parameters.
Order book	A table of pooled quantities of anonymous orders, ordered by a price.
Trade history	A table of prices and amounts sorted by time for a specific trading pair..
Market order	An order to be executed immediately over orders from the order book. Immediacy of an execution is a priority to price.
Limit order	An order to be located into the order book. The limit order provides liquidity to the market.
Maker order	A limit order to add to the order book. It supplies liquidity to the market.

Taker order	An order tries to exploit momentum market liquidity. If not matched, it is processed as was defined by an owner: to cancel or to add to the limit order book.
POST-only order	An order providing market liquidity. It exists only as a market-maker order, available for decision making of market participants.
Good-til-canceled (GTC)	An order with maturity. Maturity is defined explicitly: canceled by a submitor, or automatically in 90 days.
Immediate or cancel (IOC)	Immediacy parameter. An order to be executed immediately or to be canceled. Allows partial execution.

II.2 System architecture

The system can be described by the following diagram.



The big arrow is initiation of a contract, updating contract facilities are not depicted. *FlexClient* contract is a front-end of the system and serves all interactions with final users' dApp, including initialization of clients, order-by-order trade. Currently automatic trade (AMM) and a withdrawal of funds are not implemented.

A client of the *FlexClient*, identified by *UserIdIndex,* may have few accounts, each identified by the *AuthIdIndex* contract. Thus every account of a client is uniquely identified by the pair (*UserIdIndex*, *AuthIdIndex*). A client has a unique *UserIdIndex* and may have multiple *AuthIdIndex*s. The first contract serves to distinguish the identity of a client, the second one to distinguish between her/his wallets.

DAO community initializes token trading pairs with the *XchgPair* contract. The contract manages tokens by assigning a unique address for every pair of tokens. Currently only the TIP-3-TIP-3 trading pair is available. Introduction of the TIP-3 - EVER pair is pending. This address is used to identify all operations of one specific token over another. One token is nominated as a means of exchange and is addressed as a major token, another is a minor respectively.

Contract *PriceXchg* serves to set up prices, collect orders and process them. A client chooses a price, amount and property of an order, which is processed by the contract. A result of the proceccessing is an execution or rejection of the order.

Traditional market order books are not stored in the system, but are recalculated every time by the *FlexClient* contract with an address of a trading pair and one address of every price.

Flex

The Flex contract supports all distributed services of the system and provides coordination for functionality. It can register and delist trading pairs while registration requests can be done by anybody and can be approved or rejected by Owner only. The delisting can be done by Owner only.

FlexClient

The *FlexClient* contract is a front-end contract to match a final client and Flex marketplace services. *FlexClient* initializes a client for Flex with a unique *UserIdIndex* contract. Operations a client is interested in are to buy or to sell, with two independently choosing options each, POST or IOC. The *FlexClient* contract serves these facilities for a client. The POST option allocates an order into a limit order book and makes it open to observe for everybody at the market. The IOC option induces an immediate execution of a transaction, or a cancellation. At least one option must be always chosen.

PriceXChg

The contract supports a price formation for the tip3/tip3 exchange. First tip3 in a pair is major and terms "sell," "buy", "amount" are related to the first TIP-3 token in the pair. Second TIP-3 is called "minor.”

Price exchange contract enables trading of a pair at a specific price. It is deployed when a user creates a buy or sell order. Its address is deterministically calculated from the trading pair and price value, thus allowing to locate a specific price contract without querying the blockchain. Multiple Price exchange contracts comprise the Order book for a trading pair.

XchgPair

A trade is organized by the contract *XchgPair* , it makes a trading pair of one token in terms of another, the central service of the FLEX.

Exchange pair contract defines a trading pair, deploys price exchange contracts and stores pair information, including notification subscription address and Price exchange code hash, which is calculated from Flex Root get method. To optimize search each trading pair has a unique Price exchange code hash by which all Price contracts of this pair may be located.

II.3. Global properties

`GFL.1`	New Flex exchange can be created by anybody (after creation he called an Owner (or DAO))
`GFL.2`	New exchange pair can be requested by anybody
`GFL.3`	New exchange pair can be registered only when it's approved by the Owner
`GFL.4`	The Owner can reject pair registration
`GFL.5`	Only the Owner can delist the Pair
`GFL.6`	In case of delisting all the deposits related to the pair to be delisted must be unlocked and returned to their owners
`GFL.7`	The Owner may require some fee for the pair registration request, pair registration and even pair rejection
`GEX.1`	Anybody can send a "sell" (to sell major token) or "buy" (to sell minor tokens) order at the specified price. The order amount must be locked
`GEX.2`	If the order has a counterorder ("buy" for "sell" and vice versa) at the same price it must be executed immediately, fully or partially.

	Execution means exchange of *amount* major tokens to $\frac{price\ numerator}{price\ denominator} \times amount$ minor tokens. If the required amount of tokens is not available at either side the maximum available amount is used. In the latter case the execution is called as partial execution
`GEX.3`	In case of a few counterparts the execution goes sequentially starting from the oldest counterpart until either the order is fully executed or all the counterparts are executed
`GEX.4`	The Owner may take a fee for order placement as well as for order execution. But execution fee for maker (counterpart) must be negative
`GEX.5`	If the order has an IOC type, then in case it's not fully executed immediately it automatically canceled and the remaining part is unlocked and returned to the order creator
`GEX.6`	Otherwise, the remaining part must be put into the order book and wait until it's fully executed by the newly created counterpart, canceled or expired.
`GEX.7`	Any order on the order book can be canceled by the order creator. Some fees can be applied.
`GEX.8`	Upon cancellation the remaining deposit must be unlocked and returned to the order creator.
`GEX.9`	The order price must be not less than some minimal limit and be even divisible by some step

Section III. Functional properties

III.1. Generic properties

Numbers

All the numbers are integers or rational. The rational numbers are represented in $\frac{numerator}{denominator}$ form. The following properties for numbers are in place for all the methods.

`NUM.1`	All the numbers are not negative
`NUM.2`	The denominators for all the rational numbers are positive

Exceptions

Exceptions can be raised due to the following reasons:

- Exceptions raised due to improper external behavior such as incorrect input parameters or insufficient balance
- Exceptions raised due to software bugs

The following requirement is in place:

`EXP.1`	All the exceptions may be raised only if they are expected by the properties below

Attack protection

While the most common attacks should be handled either by the blockchain itself or by business logic of smart contracts the replay attack[7] is to be handled separately. While some of the *Everscale* compilers such as *EverX Solidity* include the defensive code automatically keeping this activity hidden, others, such as *EverX C++*, still require some explicit actions from the developer, keeping the related boilerplate as minimal, though. The property below reflects the requirement to be secure from such attacks.

`ARP.1`	All the smart contracts must be protected against Replay attack

Gas policy

`GAS.1`	All the methods than can be invoked externally must have *ACCEPT* primitive to be invoked before 10 000 of gas is consumed
`GAS.2`	After ACCEPT invoking no exception throwing is allowed under any circumstances[8].

III.2. Flex

The *Flex* contract represents the stock exchange itself.

[7] https://www.certik.com/resources/blog/blockchain-replay-attack

[8] If any method fails with an exception (with default handler) no blockchain record is created that means that the next validator will take it for execution (writing off the gas cost from the callee contract), then the next one etc. Effectively, it means that the callee contract will pay the cost a few hundred times that will likely lead to the exhausting the balance

State

The following data describes the state of *Flex* contract:

Item	Subitem	Description
Public key		The public key of the deployer
Owner address		The address of the deployer, if internal, otherwise, *null*
Ever config		
	TIP-3 Transfer	Cost of TIP-3 transfer
	Return ownership	Cost of TIP-3 ownership return
	Deploy pair	Cost of *XchgPair* deployment
	Order answer	Cost of sending an *Order* answer
	Process queue	Cost of queue processing
	Send notify	Cost of sending processing notification
Listing config		
	Register wrapper cost	Funds required (from client) to deploy wrapper
	Reject wrapper cost	Funds to be taken in case of rejected wrapper. The rest will be returned.
	Wrapper deploy value	Funds to be sent in wrapper deploy message

	Wrapper keep balance	Wrapper will try to keep this balance (should be less than *Wrapper deploy value*)
	External wallet balance	Funds to be sent in external wallet deploy message
	Reserve wallet value	Funds to be sent with reserve wallet deploy in Wrapper
	Pair register cost	Funds required to be sent by client to deploy xchg/trading pair
	Pair reject cost	Fee to be taken from the client in case of rejecting pair registration
	Pair deploy value	Funds to be sent in pair deploy message
	Pair keep balance	Pair must try to keep this balance
	Register return value	Value to be sent with registration answer callback
Code of contracts		
	XChgPair code	Code of the *XChgPair* contract
	Wrapper code	Code of the Wrapper contract
	External wallet code	Code of the external wallet contract
	Flex wallet code	Code of the *Flex* wallet contract
	XChgPrice code	Code of the *XChgPrice* contracts

	EVER Wrapper code	Code of the EVER Wrapper contract

Extra definitions

Owner

In case the contract was internally created, the *Owner* is a contract represented by *Owner address*, otherwise - the participant represented by *Public key*.

Properties

Construction

`FXC.1`	The following state attributes must be provided as input parameters: ● *Public key* ● *Owner address* ● *Deals limit* ● *Evers config* ● *Listing config*
`FXC.2`	The contract can be created by anybody, without restrictions
`FXC.3`	*Pair register cost* must be not less than *Pair reject cost* + *Register return value*, otherwise *costs_inconsistency* exception must be thrown

Code setting

Before *Flex* becomes fully functional the *Code of contracts* attributes must be set. The following requirements must be met:

FXS.1	*Code of contracts attributes* can be set by *Public key* owner only, otherwise *sender_is_not_deployer* exception must be thrown
FXS.2	*Code of contracts attributes* can be set only once, otherwise *cant_override_code* exception must be thrown
FXS.3	*XChgPair code* cannot be set without prior setting of *XChgPrice code*, otherwise *xchg_price_code_undefined* exception must be thrown
FXS.4	*XChgPair code* must be properly salted to get pairs intended to be used with different *Flex* instances as well with different *TIP-3* coins to be distinguished (to have different addresses)
FXS.5	*XChgPair code*, *Wrapper code, EVER Wrapper code, XChgPrice code* must have two references[9] in a code cell, otherwise, *unexpected_refs_count_in_code* must be thrown

Transferring

Flex must have a possibility to transfer money both from itself (in case of EVERs) and from the reserve wallet of the provided Wrapper (in case of tokens) to an arbitrary address.

FXT.1	Transferring can be done by *Owner* only, otherwise *sender_is_not_my_owner* exception should be thrown
FXT.2	All the parameters of transfer must correspond to the respected input parameters

[9] It's supposed that the reader gets acquainted with the concept of cells and "All the data is a cell" paradigm. Otherwise, please refer to the TVM documentation (see, section 3.1)

FXT.3	In case of transferring tokens from Wrapper the answer address should be set to *Owner address* if exists or kept empty otherwise

Deploy pair

To deploy a pair at first it should be requested (by anybody). Then, the owner can either accept or reject this request.

FXP.1	On approval *Pair Deploy value* must be larger must be larger than *Pair keep balance* otherwise the deployment of pair fails
FXP.2	Message value for the pair registration request must be more than *Pair Register Cost* added by *Register return value*, otherwise exception *not_enough_funds* must be thrown
FXP.3	Until the request is either accepted or rejected the second request with the same public key is not allowed, otherwise *xchg_pair_with_such_pubkey_already_requested* exception must be thrown
FXP.4	All the *XchgPair* parameters must be provided as input parameters of the pair registration request
FXP.5	*Minimal amount* and *Minimal move* for the requested pair must be positive, otherwise, *incorrect_config* exception must be thrown
FXP.6	Callback message for the pair registration request should have *Register return value* as a value and address of the potential *XchgPair* as a payload
FXP.7	Pair registration request may be sent by any contract
FXP.8	Pair approval and rejection can be sent by *Owner* only, otherwise *sender_is_not_my_owner* exception should be thrown

FXP.9	If the Flex is internally created (*Owner Address* exists) all the unused part of the value must be returned back after approval or rejection
FXP.10	If the pair registration request with the specified public key was either already handled or never sent *xchg_pair_not_requested* exception must be thrown upon approval or rejection
FXP.11	If the pair registration request does not throw an exception it must deploy an XchgPair contract specified by request parameters. The details of deployment are described below.
FXP.12	If the pair deployment was successful the initiator of pair registration must be informed while all the value sent at the time of registration request but *Register return value* added by *Pair register cost* must be returned
FXP.13	If the pair rejection was successful the initiator of pair registration must be informed while all the value sent at the time of registration request but *Register return value* added by *Pair reject cost* must be returned

Wrapper registration

The present section describes the registration of Wrappers discussed in the TIP-3 section. The request (both in the form of a generic Wrapper or specific EVER Wrapper) can be sent by any contract (it must be internal), while approval or rejection can be done by *Owner* only. The following properties are in place:

FXW.1	Wrapper registration may be requested by any contract
FXW.2	The message value for the wrapper registration request must be greater than *Register wrapper cost* added by *Register return value*, otherwise *not_enough_funds* exception must be thrown

`FXW.3`	Until the request is either accepted or rejected the second request with the same public key is not allowed, otherwise *wrapper_with_such_pubkey_already_requested* exception must be thrown
`FXW.4`	All the Wrapper parameters must be provided as input parameters of the wrapper registration request
`FXW.5`	Callback message for the wrapper registration request should have *Register return value* as a value and address of the potential Wrapper as a payload
`FXW.6`	Wrapper approval and rejection can be sent by *Owner* only, otherwise *sender_is_not_my_owner* exception should be thrown
`FXW.7`	If the Flex is internally created (*Owner Address* exists) all the unused part of the value must be returned back after approval or rejection
`FXW.8`	If the Wrapper registration request with the specified public key was either already handled or never sent *xchg_pair_not_requested* exception must be thrown upon approval or rejection
`FXW.9`	If the Wrapper registration request does not throw an exception it must deploy an Wrapper contract specified by the request parameters. The details of deployment are described at the corresponding section.
`FXW.10`	If the Wrapper deployment was successful the initiator of wrapper registration must be informed while all the value sent at the time of registration request but *Register return value* added by *Register wrapper cost* must be returned
`FXW.11`	If the Wrapper rejection was successful the initiator of Wrapper registration must be informed while all the value sent at the time of registration request but *Register return value* added by *Wrapper reject cost* must be returned

III.3. FlexClient

This contract represents the client of the *Flex*.

State

State of the *FlexClient* contract can be described by the following attributes.

Owner	Owner (its public key and workchain id) of the *FlexClient* contract, in other words, its creator
Item	**Description**
Flex	The *Flex* contract the *FlexClient* is attached to
Flex wallet code	Code of the Flex wallet contract
AuthIndex code	Code of the *AuthIndex* contract
UserIDIndex code	Code of the *UserIDIndex* contract

Properties

Construction

`FCC.1`	At construction the not-null *Owner* must be provided as an input parameter, otherwise *zero_owner_pubkey* exception must be thrown

Code and *Flex* setting

The present section describes the properties related to code setting. The setting of all the *code*-attributes is required before the *FlexClient* can be used in a full-scale mode, however these attributes can be set by the *Owner* only. The same is about the *Flex* attribute that is also required to be set for the correct performance of the *FlexClient*.

`FCS.1`	All the code setting and *Flex* changing operations can be initiated by the *Owner* only, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCS.2`	All the codes as well as *Flex* must correspond to the input parameters
`FCS.3`	*AuthIndex code* must be properly salted to achieve distinguishing of addresses designated for different *Flex*'s
`FCS.4`	*UserIdIndex code* can be set only when *AuthIndex code* has been set before, otherwise *auth_index_must_be_defined* exception must be thrown
`FCS.5`	*UserIdIndex code* must be properly salted to achieve distinguishing of addresses designated for *Owner*s and *AuthIndex code*s

Ownership lending

The *FlexClient* can lend (as well as relend) an arbitrary amount of TIP-3 tokens to the arbitrary address. This operation can be done by the *Owner* only. The details of token lending are described in the corresponding section.

`FCL.1`	The lending is possible by *Owner* only
`FCL.2`	All the parameters of lend are provided as input parameters
`FCL.3`	In case of success TIP-3 lend is performed

Exchange order cancel

The *Owner* should be able to cancel any order he is eligible for. The cancellation itself is made by *PriceXchg* contract.

`FCX.1`	The cancellation can be initiated by *Owner* only
`FCX.2`	The request to cancel must be sent to the proper *PriceXChg* contract with parameters corresponded to the input parameters of the request

Transfer

The *FlexClient* can transfer EVERs from its own account as well as TIP-3 tokens from the arbitrary one. The activity being discussed can be initiated by the *Owner* only.

`FCT.1`	Any kind of transfer can be initiated by the *Owner* only
`FCT.2`	All the details of the transfer must be provided as input parameters

Price exchange deployment

`FCP.1`	*PriceXChg* deployment can be done by *Owner* only, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCP.2`	All the *PriceXChg* parameters must be provided as input parameters
`FCP.3`	The price (as an input parameter) must be positive, otherwise *zero_num_in_price* exception must be thrown
`FCP.4`	The required (provided as an input parameter) amount of TIP-3 tokens must be lent to the *FlexClient*

Wrapper and XchgPair registration

Proxy functionality that runs the corresponding functionality of Flex.

`FCW.1`	Only *Owner* can request Wrapper and *XchgPair* registration, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCW.2`	All the parameters of the object to be registered are provided as input parameters
`FCW.3`	The corresponding functionality of *Flex* must be called

Empty Flex wallet deployment

`FCE.1`	Only *Owner* can deploy an empty Flex wallet, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCE.2`	*Flex wallet code* must be set, otherwise *missed_flex_wallet_code* exception must be thrown
`FCE.3`	In case of no exceptions mentioned above, new wallet must be created while its data must be provided as input parameters
`FCE.4`	Depending on the input parameters the newly created wallet may be lent to the specified contract

Index management

Owners can create, relend and destroy *UserIDIndex* contracts.

`FCI.1`	Only *Owners* can create, relend or destroy *UserIdIndex* instances, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCI.2`	*UserIdIndex code* must be set to deploy an index, otherwise *missed_user_id_index_code* exception must be thrown
`FCI.3`	Upon request to create, relend or destroy UserIdIndex the corresponding methods of *UserIdIndex* or TIP-3 Wallet must be called with the arguments provided in the original request

Wallet management

Owners can burn and set trade restrictions on TIP-3 wallets.

`FCM.1`	Only *Owners* can burn and restrict, otherwise *message_sender_is_not_my_owner* exception must be thrown
`FCM.2`	Upon request to burn or restrict wallets the corresponding methods of TIP-3 Wallet must be called with the arguments provided in the original request

III.4. XchgPair

This contract represents the trading pair.

State

The following data describes the state of the pair.

Item	Description
Major coin	The TIP-3 coin that is named as a major
Minor coin	The TIP-3 coin that is named as a minor
Minimal amount	The minimal amount of *Order* to be placed
Minimal move	Any amount to be placed must be a even divisible by this attribute
Price denominator	The price of the *Minor Coin* in terms of *Major coin* must be considered as $\frac{numerator}{denominator}$. The *numerator* is defined by the present attribute, while the *denominator* is defined by *PriceXchg* contract.
Notify address	An address where all the changes must be notified to

Properties

Deployment

The following properties must be in place when the trading pair is being deployed.

`XPD.1`	It should be no possibility for double deployment, otherwise, *double_deploy* exception should be raised
`XPD.2`	The *Minimal amount* must be provided and must be positive
`XPD.3`	The pair can be deployed by *Flex* only, otherwise the following exceptions are possible: ● *bad_incoming_msg* ● *unexpected_refs_count_in_code* ● *only_flex_may_deploy_me*
`XPD.4`	*Major coin* must be provided as an input parameter
`XPD.5`	*Minor coin* must be provided as an input parameter
`XPD.6`	*Minimal move* must be provided as an input parameter
`XPD.7`	*Price denominator* must be provided as an input parameter
`XPD.8`	*Notify address* must be provided as an input parameter
`XPD.9`	All the **EVERs** over the amount specified by the input parameter must be returned

III.5. PriceXchg

The *PriceXchg* contract represents all the active orders for the particular *Flex* and pair of tokens (one of them is called major and another is minor) at the same price.

State

Item	Description
Config	*Ever config* attributed of the host *Flex* combined with state of the host *XchgPair*
Price numerator	The proposed price of minor token in major token can be represented as $\frac{numerator}{denominator}$ form. The current attribute represents the *numerator* while the *denominator* can be taken from the *Config*'s *Price denominator* attribute described above.
Sell order book	The queue (FIFO) of all the sell orders at the same price (defined by *Price numerator*)
Buy order book	The queue (FIFO) of all the buy orders at the same price (defined by *Price numerator*)

Each *Order* in the *Buy Order book / Sell order book* has the following attributes:

Item	Description
isPostOrder	Indicates if the order should be enqueued in case no immediate buyers/sellers are found for the whole lot
Amount	Remaining amount of major tokens to be bought or sold
Token provider	The TIP-3 wallet of the *FlexClient* that has tokens to be sold
Client	The *FlexClient* that submitted the *Order*

Finish time	The time when the order becomes expired

Special definitions

Expired *Order*

The *Order* is called expired if its *Finish time* is bigger or equal than *NOW*[10].

Minor amount

For the *Order* the *Minor amount* is $Order\ amount \cdot \frac{Price\ numerator}{Config's\ Price\ denominator}$

Counterpart

For the *Order*s from *Sell order book* it's a set of *Orders* from *Buy order book*, and vice versa.

Transaction processing (*sell* as a parameter)

- Happens between the *Order* from *Order book* and his earliest *member of Counterpart* (*CounterOrder*)
- If there is no *CounterOrder* nothing happens, but in case of no *isPostOrder* the *Order* is not added to the corresponding *Order book*
- If *sell*
  - *Amount* of *CounterOrder* exceeds the *Amount* of *Order*, then:
    - *Amount* of *Order* is transferred from *Major receiver* of *Order* to the *Major receiver* of *CounterOrder*

[10] NOW is a physical time as defined by the corresponding TVM primitive

        - *Minor amount* of *Order* is transferred from *Minor receiver* of *CounterOrder* to the *Minor Receiver* of *Order*
        - Message value of transfer is *Config*’s *TIP-3 Transfer*
        - *Order* is removed from the *Sell order book*
        - *Amount* of *CounterOrder* is decreased by *Amount* of *Order*
        - If *Amount* of *CounterOrder* becomes zero then *CounterOrder* is removed from *Buy order book*
    - Otherwise:
        - *Amount* of *CounterOrder* is transferred from *Major receiver* of *Order* to the *Major Receiver* of *CounterOrder*
        - *Minor amount* of *CounterOrder* is transferred from *Minor receiver* of *CounterOrder* to the *Minor Receiver* of *Order*
        - Message value of transfer is *Config*’s *TIP-3 Transfer*
        - *CounterOrder* is removed from the *Buy order book*
        - *Amount* of *Order* is decreased by *Amount* of *Order*
- Otherwise:
    - *Amount* of *Order* exceeds the *Amount* of *CounterOrder*, then:
        - *Amount* of *Order* is transferred from *Major receiver* of *CounterOrder* to the *Major receiver* of *Order*
        - *Minor amount* of *Order* is transferred from *Minor receiver* of *CounterOrder* to the *Minor Receiver* of *Order*
        - Message value of transfer is *Config*’s *TIP-3 Transfer*
        - *Order* is removed from the *Buy order book*
        - *Amount* of *CounterOrder* is decreased by *Amount* of *Order*
        - If *Amount* of *CounterOrder* becomes zero then *CounterOrder* is removed from *Sell order book*
    - Otherwise:
        - *Amount* of *CounterOrder* is transferred from *Major receiver* of *CounterOrder* to the *Major Receiver* of *Order*

- *Minor amount* of *Order* is transferred from *Minor receiver* of *Order* to the *Minor Receiver* of *CounterOrder*
- Message value of transfer is *Config*'s *TIP-3 Transfer*
- *CounterOrder* is removed from the *Sell order book*
- *Amount* of *Order* is decreased by *Amount* of *Order*

Order processing

The *Order processing* is a sequence of:

- A continuous repetition of *Transaction processing (true)* until it becomes constant
- A continuous repetition of *Transaction processing (false)* until it becomes constant

Properties

Initialization

`PXI.1`	*Config* and *Price numerator* must be initialized by the values provided by the initiating *FlexClient*
`PXI.2`	*Sell order book* and *Buy order book* must be initialized as empty list
`PXI.3`	During the handling the message tree based on the same external trunk[11] at least one *Order* must be placed into either *Sell order book* or *Buy order book*
`PXI.4`	*Price numerator* must be a even divisible by *Config*'s *Minimal*

[11] As the ecosystem being described does not have an access to exotic system stuff such as toc-tocs, all the activity is initiated by external external messages, while the handlers can issue their own internal messages, and their handlers … Thus, we get some message tree based on the original external message as a trunk.

	move, otherwise *incorrect_price exception* must be thrown during *Order placement*

Order placement

PXP.1	Order placement can be done by and must be automatically initiated as a callback to TIP-3 lending. Can be initiated manually by anybody
PXP.2	The message value of the order placement request must be not less than sum of the following *Config*'s attributes: ● *Process queue* ● Tripled *TIP-3 Transfer* ● *Send notify* ● *Return ownership* ● *Order answer* Otherwise, the callback message with error named *not_enough_tons_to_process* must be sent
PXP.3	If the TIP-3 wallet provided as an input parameter does not correspond to *Token Provider* the callback message with error named *unverified_tip3_wallet* must be sent
PXP.4	The provided amount of the *Order* must be not less than *Config*'s *Minimal amount*, otherwise the callback message with error named *not_enough_tokens_amount* must be sent
PXP.5	In case of selling, the amount of tokens lent must be not less than the provided amount of the *Order*, otherwise the callback message with error named *too_big_tokens_amount* must be sent
PXP.6	In case buying, the amount of tokens lent must be not less than the

	provided *Minor amount* of the *Order*, otherwise the callback message with error named *too_big_tokens_amount* must be sent
`PXP.7`	If the *Order* is *Expired* the callback message with error named *expired* must be sent
`PXP.8`	If any error described above occur the *Order* - related lends must be returned but *Return Ownership Config*'s value
`PXP.9`	In case of no errors the *Config*'s *Notify address* must be informed about the order placement
`PXP.10`	In case of no errors the new *Order* must be placed with immediate handling: ● The *Order* (defined by input arguments) is added to either *Sell order book* or *Buy order book* (depending on input arguments) ● The *Order* is immediately processed by *Order Processing* story
`PXP.11`	If the both *Sell order book* and *Buy order book* become empty the *PriceXchg* contract must suicided.
`PXP.12`	All the unused gas must be returned to the initiator
`PXP.13`	If the *Order* has the attribute *isPostOrder* and is not fully processed the the remaining part must be moved to the corresponding Order Book, otherwise the remaining part must be unlocked and returned to the order creator

Order canceling

`PXC.1`	Can be invoked by the *Client* only

PXC.2	In case of no errors the *Order* must be removed from either *Sell order book* or *Buy order book* depending on input parameters
PXC.3	If the both *Sell order book* and *Buy order book* become empty the *PriceXchg* contract must suicided.
PXC.4	All the unused gas must be returned to the initiator
PXC.5	All the ownership related to *Order* being canceled must be returned
PXC.6	Upon cancel the *Client* must be informed
PXC.7	If the both *Sell order book* and *Buy order book* become empty the *PriceXchg* contract must suicided.
PXC.8	All the unused gas must be returned to the initiator

III.6. UserIdIndex

Properties

Deployment

UID.1	Only *FlexClient* can deploy *UserIdIndex*, otherwise *only_client_may_deploy_me* exception must be thrown
UID.2	Double deployment is forbidden, otherwise *double_deploy* exception must be thrown
UID.3	In case of no exception new *AuthIdIndex* must be deployed with the attributes provided by *FlexClient*

Relending and destroying AuthIdIndex

`UIR.1`	Relending and destroying *AuthIdIndex* can be done by *FlexClient* only, otherwise *message_sender_is_not_my_owner* exception must be thrown
`UIR.2`	Relending is a subsequent process of removing the old *AuthIdIndex*, if exists and deploying a new one

III.7. AuthIdIndex

`AID.1`	The contract can be destroyed only by the contract that created it

Section IV. TIP-3

TIP-3 is located at a lower architectural level than Flex.

IV.1. Summary

TIP-3 is a custom token built upon the Everscale network. It can be transferred from one TIP-3 wallet to another, acting as an independent currency (just based on Everscale blockchain), but also offers some additional functionality such as allowance (ability to automatically write off up to the specified amount of tokens) and lending (moving some tokens into an external custody without ability to cancel it until lending is finished by time out or by explicit request from lendee).

IV.2. Business-level description

TIP-3 has separate storage for each user’s contract. An instance of a token starts from a root token wallet, which deploys a contract. The root contains a minting function, a number of granted tokens and a total amount of tokens. Access to the root token wallet is controlled by an owner. A client may have more than one TIP-3 token wallets.

TON Token Wallet (TTW) is initialized with a root token wallet, a root public key, a wallet public key and a zero balance. A TON public key can be deployed only by an owner of a root token wallet.

The TTW has guards in respect of available amounts of transferable funds, well-defined target contracts, and error processing from unsuccessful transfer operations.

TTW supports such features as allowance and lending. Allowance (can be compared to card subscription in fiat world) is a permission to the external contract to spend the fixed amount of tokens. Such a permission can be revoked at any moment.

Lending is also a permission to the external contract to spend the fixed amount of tokens but it can not be revoked by the lender. Instead, it can be released by timeout or by explicit request from lendee. Thus, it can be compared with a security deposit.

Surprisingly, TTW also can create and cancel Flex orders. In our opinion, it’s a mistake on architecture, lower level should not directly use higher one.

The contract Wrapper TIP-3 Root interacts directly with a client (FlexClient). It accepts and converts external tokens into Flex TIP-3 tokens, deploys Flex TIP-3 wallets, stores relevant token information, including Flex TIP-3 wallet code hash.

IV.3. Properties

Root

Attributes

Item	**Description**
Name	Token name
Symbol	Token symbol (such as BTC)
Decimals	The decimal logarithm of “main” token (say 1 EVER = 10^9 nanoEVER), so its *Decimals* is 9
Owner	The contract or public key provider who created the contract
Total supply	The total amount of issued tokens
Wallet code	The code of *Wallet*

Properties

`RTC.1`	*Owner* and other attributes must be set at deployment
`RTC.2`	*Wallet code* must be set before starting using the *Root*

RTC.3	*Wallet code* can be set only once, otherwise *cant_override_wallet_code* exception must be thrown
RTC.4	*Wallet code* hash must correspond to the predefined (at compile time) value, otherwise *wrong_wallet_code_hash* exception must be thrown
RTC.5	*Wallet code* can be set by *Owner* only, otherwise *message_sender_is_not_my_owner* exception must be thrown
RTC.6	In case of internal requests all the unused gas must be returned to the sender
RTC.7	New *Wallet* with some initial tokens granted can be deployed by the *Owner* only, otherwise *meesage_sender_is_not_my_owner* exception must be thrown
RTC.8	*Wallet*, on deployment, may be provided with some tokens (the number is to be defined by the *Owner*), however, the total number of granted tokens (for all the deployments) should not exceed *Total Supply*, otherwise *not_enough_balance* exception must be thrown
RTC.9	*Wallet* must be deployed with either public key or contract address, called *Wallet Owner*, otherwise *define_pubkey_or_internal_owner* exception must be thrown
RTC.10	New empty *Wallet* can be deployed by anybody
RTC.11	A *Wallet* may be provided with some tokens (the number is to be defined by the *Owner*), however, the total number of granted tokens (for all the deployments) should not exceed *Total Supply*, otherwise *not_enough_balance* exception must be thrown. This request can be done by *Owner* only, otherwise *meesage_sender_is_not_my_owner* exception must be thrown

RTC.12	The *Owner* can mint new tokens, thus increasing *Total Supply*. This request can be done by *Owner* only, otherwise *meesage_sender_is_not_my_owner* exception must be thrown

Wrapper

Attributes

Item	**Description**
External wallet	For TIP-3 wrappers only, external *Wallet* being wrapped
Reserve wallet	EVER wallet
Internal wallet code	Code of the internal wallet
Token name	*Name* of the wrapped TIP-3 token
Token symbol	*Symbol* of the wrapped TIP-3 token
Token decimals	Decimals of the wrapped TIP-3 token
Token total	*Total supply* of the wrapped TIP-3 token
Flex	Address of the *Flex* that created the *Wrapper*

Properties

Initialization

`WRI.1`	The Wrapper can be initialized only once, otherwise *cant_override_external_wallet* exception must be thrown
`WRI.2`	All the attributes must be provided as input parameters
`WRI.3`	*Internal wallet code* hash must correspond to the predefined value, *otherwise wrong_wallet_code_hash* exception must be thrown
`WRI.4`	In case of no exceptions, a new *Wallet* (not related to the state attribute, let's call it *Custodian Wallet*) must be deployed using the attributes provided
`WRI.5`	All the unused gas must be returned to *Flex*

Custodian Wallet deployment

`WRD.1`	New *Custodian Wallet* can be created by anybody
`WRD.2`	All the attributes of the new *Custodian Wallet* must be provided as input parameters
`WRD.3`	All the unused gas must be returned to the sender

Transfer handling

`WRT.1`	Must be invoked when either: ● TIP-3 tokens are received by *Wallet* (for TIP-3 *Wrapper*) ● EVERs are received by user wallet
`WRT.2`	A new *Custodian Wallet* must be deployed with attributes an tokens provided

Burning

`WRB.1`	If amount to burn is more than amount of tokens provided then *burn_unallocated* exception must be thrown
`WRB.2`	Only the owner of the External wallet can initiate the burning

Wallet

State

Item	**Subitem**	**Description**
Owner		The contract or public key that created the *Wallet* or was introduced as a *Owner* by *Root*
Common items		
	Name	Name of the TIP-3 token
	Symbol	Symbol of TIP-3 token
	Decimals	Decimals of TIP-3 token (see there)
	Root	TIP-3 *Root*
Wallet balance		TIP-3 *Wallet* balance
Allowance		

	Spender	Allowance beneficiary
	Remaining tokens	Number of TIP-3 tokens still allowed to the beneficiary
Lendees		
	Lendee	The particular lendee
	Lend balance	Lent amount
	Finish time	Lend finish time
	Restrictions	Indicates for what *Flex* and what price the orders can be handled

Properties

Deployment

`TWD.1`	All the *Common Items* must be set at deployment
`TWD.2`	If the Wallet is deployed by Root it can get some initial tokens as a *Wallet balance*

Transfer

`TWT.1`	Transfer can be initiated by *Owner* only
`TWT.2`	Transfer can be performed by: ● *Owner* in case either:

	○ No lending is in place ○ Amount of not lent tokens exceeds the required amount and the configuration allows to transfer by *Owner* while having lending ● *Lendee* in case *Lend balance* exceeds the required amount Otherwise, *message_sender_is_not_my_owner* or *not_enough_balance* exception must be thrown
TWT.3	Transfer must be accompanied by the enough EVERs as defined by configuration, otherwise not_enough_evers_to_process exception must be thrown
TWT.4	Transfer can be done either to existing TIP-3 wallet or to newly deployed one
TWT.5	Upon the transfer the balance must be decreased by the required amount and the destination balance must be increased by the same amount

Minting

TWM.1	Minting can be initiated by *Root* only
TWM.2	Upon minting the *Wallet* balance is increased by the amount provided

Destroying

TWR.1	Destroying can be called if the *Wallet balance* is zero, otherwise

	destroy_non_empty_wallet exception must be thrown
TWR.2	Can be initiated by *Owner* only, otherwise *message_sender_is_not_my_owner* must be thrown
TWR.3	All the gas must be returned to the destination provided as an input parameter

Burning

TWB.1	Can be invoked only if the *Root* is a *Wrapper*
TWB.2	Can be initiated by *Owner* only, otherwise *message_sender_is_not_my_owner* must be thrown
TWB.3	Must invoke burning implemented by the *Wrapper*

Lending

TWL.1	Can be initiated by *Owner* only, otherwise *message_sender_is_not_my_owner* must be thrown
TWL.2	The provided finish time (as an argument) must be greater than NOW, otherwise *finish_time_must_be_greater_than_now* exception must be raised
TWL.3	The *Lend amount* can be increased and *Finish time* can be changed without canceling the current lending
TWL.4	No *Allowance* is permitted, otherwise *allowance_is_set* exception must be thrown

`TWL.5`	Upon completion the operation the *Lendee* can operate with *Lend Amount* until *Finish time* or explicit ownership return

Orders

DISCLAIMER!!! While TIP-3 is located at a lower level than Flex, the placement of Flex functionality here is considered as an architecture error by the authors of the present document. However, the low level specification is provided

`TWO.1`	To create an order *Lending* must be set and *Finish time* must be greater than NOW, otherwise *finish_time_must_be_greater_than_now* exception must be thrown
`TWO.2`	To create an order *Lend balance* must be positive, otherwise *zero_lend_balance* exception must be thrown
`TWO.3`	To create an order *Restrictions* must be set and the provided input must fit them, otherwise *restriction_not_set*, *wrong_flex_address* or *wrong_price_xchg_code* exceptions must be thrown
`TWO.4`	By "Create an order" request, in case of no exceptions, a new *PriceXChg* contract must be created, with the parameters provided from input
`TWO.5`	An order can be created or canceled by either Owner or Lendee, otherwise *message_sender_is_not_my_owner* must be thrown

Section V. Possible issues

V.1. Business-level issues[12]

Currently the system offers all basic services for order-by-order trade. Development of automated trade is pending. Every buy/sell operation chosen by a client is accompanied by a reservation of an adequate amount of a respective token along with reservation of execution fee. This excludes the states of default of a buyer and a seller after an operation.

The system has the following features, important for trade.

- A minimum tick size for the major token. This parameter has an impact on market volatility.
- A lower bound for the value of a major token. This prevents division by zero for a means of trade.

Nevertheless the system is exposed for traditional vulnerabilities of stock exchanges.

Sharp market fall. Experiences of many traditional exchanges demonstrate that markets are vulnerable to hurdle behavior on a falling market. Traditional management tool is a temporary termination of trades, to let the market participants cool down expectations. Stock exchanges can perform this instantly, but DAO administration can not perform this operation rapidly. A possible solution could be an implementation (or incorporation) of a voting system to make urgent decisions in such cases.

[12] The present analysis was done by PhD in Economics(both Russian and EU(Ca Foscari, Venezia)), expert in microfoundations of macro and monetary economics as well as in game theory. The findings can be disputable by the community and require additional non-technical sessions with interessants.

SEC documents[13] provide definitions for several kinds of market manipulations and supply recommendations to monitor them. These market failures may emerge at the market.

Arbitrary Quotes. This is the existence of the orders which significantly differ from others orders coming regularly from the same indice pair (*UserIdIndex*, *AuthIdIndex*). Usually such activity is evaluated as supplying false market signals by a client, which can be uniquely identified. Actions for such cases should be regulated by DAO decisions.

Wash Sales. Almost simultaneous submission of orders to both sides of the market, to buy and to sell, involving no change of beneficial ownership of the stock. This means that the same (*UserIdIndex*, *AuthIdIndex*) contract operates simultaneously at both sides of the market. Actions for such cases should be regulated by DAO decisions.

Matched Orders Placing buy or sell orders for with the knowledge that sell or buy orders of the same amount and price will be placed simultaneously

- Engaged in for the purpose of creating a false appearance of active trading in the market for security, especially by from the same (*UserIdIndex*, *AuthIdIndex*) contract.
- May be done at successively higher and higher prices to move market price of security upwards.

Actions for such cases should be regulated by DAO decisions.

Domination and Control of Market The situation emerges when a market participant tries to dominate and control the market for a particular security, for example by holding a dominant portfolio. After control is established, the participant, or a group of affiliated participants is free to arbitrarily move bid and

[13] For example: https://www.sec.gov/files/Market%20Manipulations%20and%20Case%20Studies.pdf.

price of security upwards/downwards without reference to forces of supply and demand. Existence of the momentum traders and ill-informed traders let the manipulation succeed. Actions for such cases should be regulated by DAO decisions.

Layering. Placing orders to the market with no intention of having them executed. This creates false market signals. This operation may become important for AMM based trade. Actions for such cases should be regulated by DAO decisions.

Affiliated participants. Control for affiliated participants trade requires additional features for the system and an approvement by DAO(?)/

Pumping/dumping market activity requires additional control features from the system and an approval by DAO.

Multimarket issues

At the moment the system is expected to become multimarket, with simultaneous trade for several token pairs. This will offer clients new arbitrage opportunities between the markets within the system and between other markets. Multimarket operations will not allow markets to stay independent any more. Thus will come a problem, how good is an interaction of different instances of the *TradingPair* contracts through the *FlexClient* contract. Thus the addition of a new market becomes not a technical exercise, but a problem of sustainability of the whole system.

V.2. Technical-level issues

While the code audit was not a part of the present activity, the team found a few issues to be presented.

Major bugs

```
Flex.cpp : void setXchgPairCode(cell code)

tvm.accept();
…
require(decoded_salt.xchg_price_code.ctos().srefs()
== 2, 88);

require after accept is not allowed
```

```
Flex.cpp : void setWrapperCode(cell code)

tvm_accept();

require(code.ctos().srefs() == 2,
error_code::unexpected_refs_count_in_code);


require after accept is not allowed
```

```
Flex.cpp : void setWrapperEverCode(cell code)

tvm_accept();

require(code.ctos().srefs() == 2,
error_code::unexpected_refs_count_in_code);
```

require after *accept* is not allowed

```
Flex.cpp : address registerXchgPair(...)

require(int_value().get() >
listing_cfg_.register_pair_cost,
error_code::not_enough_funds);
…
int_return_value(listing_cfg_.register_return_value.g
et());

If message value is less than register_pair_cost +
register_return_value, exception will be thrown
```

```
Flex.cpp : std::pair<address, DFlex::xchg_pairs_map>
approveXchgPairImpl(...)

require(!!opt_req_info,
Flex<true>::error_code::xchg_pair_not_requested);

As this function is called by approveXchgPair(...)
where tvm_accept() was called before this call,
require after accept bug takes place
```

```
PriceXchg.cpp OrderRet onTip3LendOwnership(...)
```

```
else if (!args.immediate_client && (is_sell ?
buys_amount_ != 0 : sells_amount_ != 0))
      err =
ec::have_other_side_with_non_immediate_client;

It’s not a software bug, as the behavior is clearly
intended but from the business logic it sounds
strange: one can not put a post order while
counterparts present.
```

Minor bugs

```
Flex.cpp : address registerWrapper(uint256 pubkey,
Tip3Config tip3cfg)

require(int_value().get() >
listing_cfg_.register_wrapper_cost,
error_code::not_enough_funds);
…
int_return_value(listing_cfg_.register_return_value.g
et());

If message value is less than register_wrapper_cost +
register_return_value, exception will be thrown
```

```
FlexClient.cpp : void deployIndex(...)

require(user_id_index_code_,
error_code::missed_flex_wallet_code);
```

```
Incorrect exception name
```

V.3. Possible threats and attacks

The following potential threats and attacks are to be considered at the later stages of the formal verification:

- Unauthorized access. To mitigate this risk throughout the present specification a lot of attention paid to the security-related properties
- Balance exhaust. An attacker can repeatedly invoke the expensive methods externally or from cheap internal methods (in case of ACCEPT). To mitigate it the verifiers must check that all the methods proper authorize the sender and provide ACCEPT only if the sender is safe
- Replay protection. This kind of attack is discussed here, and luckily C++ SDK has macroses that allow to eliminate this risk. However, verifiers should check if these macroses were applied everywhere it's needed
- Incorrect arithmetics and logic, in particular:
  - Incorrect conversion from minor to major tokens, and vice versa
  - Inconsistency between sent and received TIP-3 tokens
  - Inconsistency between total supply of tokens and their real total supply
  - Moving GTC orders into incorrect order book
- Incorrect fee payment and change return. To mitigate this risk the present specification pays special attention to the related properties.
- Moving the internal parameters into an incorrect state. In particular:
  - Negative values
  - Frozen locked deposits
- Incorrect address calculation. In this case the price and pair exchanges can become unreachable. Despite, these properties are too

implementation-specific even for Section III, the verifiers of Stage II should pay special attention to this risk

- Meeting the business risks . Business risks should be analyzed and, possibly, included into the scope of Phase 2 or Phase 3