CHARON Exchange

Introduction

This whitepaper describes architecture and principles of work for decentralized exchange (DEX) **Charon Exchange**.

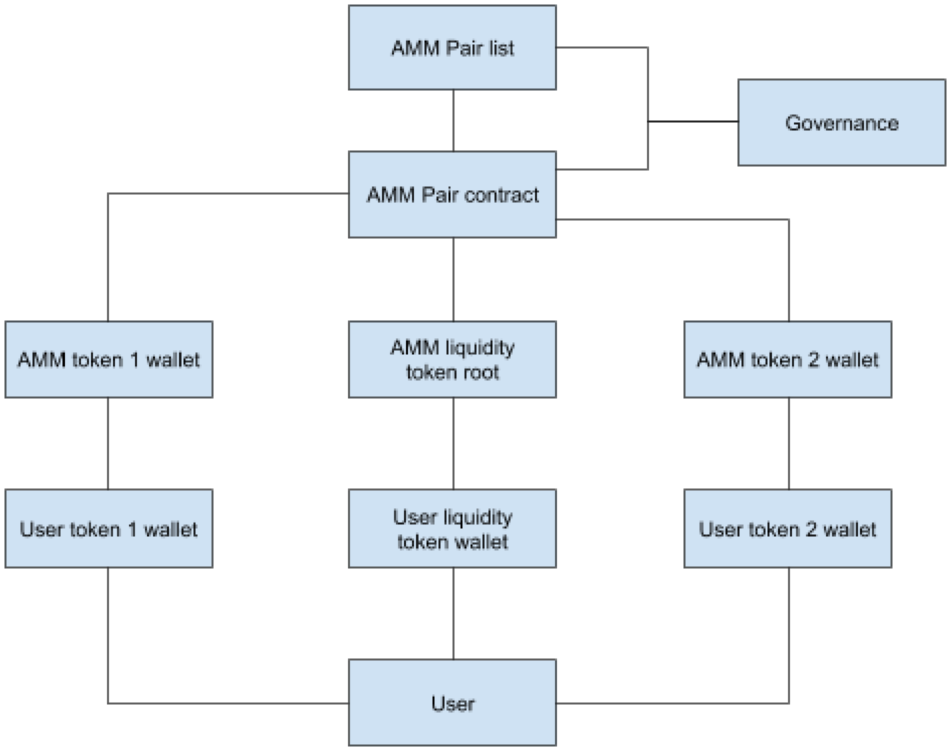
Charon Exchange uses an automated market-maker mechanism (AMM) that allows providers of liquidity deposit their funds to get profits and offers for users fast and safe exchange service with low commissions.

Charon Exchange uses the TIP3 wallet standard with modifications which are required for automatic operating and control by DEX on-chain.

Also there are described protection mechanisms from *“impermanent loss”* and *“front-running”* atack.

The example of implementation modified TIP-3, exchange smart-contract and tests for checking main features can be found at github.

Architecture of DEX



Pic. 1. Architecture of DEX

Key components of DEX (Pic. 1):

1. **AMM Pair Contract** - main smart-contract for token exchanging and providing liquidity.
2. **AMM token 1 wallet and AMM token 2 wallet** - TIP-3 wallets for storing liquidity of pair tokens. AMM Pair Contract controls them on-chain.
3. **AMM liquidity token root** - TIP-3 root for minting and burning tokens of liquidity.
4. **User liquidity token wallet** - user’s TIP-3 wallet for liquidity token.
5. **AMM Pair list** - contract contains list of all AMM Pair contracts.
6. **Governance contract** - contract controls specified parameters of DEX.

Workflow

1. User creates an external message with a list of commands to own the **TIP-3 wallet**.
2. User’s wallet sends commands, tokens and a few amount of TON Crystal to the **AMM token wallet**.
3. **AMM token wallet** sends notification about received tokens, commands, user’s data and rest TON Crystals to **AMM Pair Contract**.
4. **AMM Pair contract** saves user’s tokens at the list of balances by public key.
5. Next, **AMM Pair contract** executes commands sequentially. Commands are presented in Chapter “Commands”.
6. In case of unsuccessful execution, the user's tokens stay at balance.
7. If the user wants only to execute commands, he has to send commands without tokens.
8. On withdrawing the user’s tokens, **AMM Pair** sends a message with the user's public key and amount tokens to AMM token wallet or AMM liquidity token root. Message contains the rest of TON Crystals.
9. **AMM token wallet or AMM liquidity token root** sends tokens to user’s wallet.

Commands

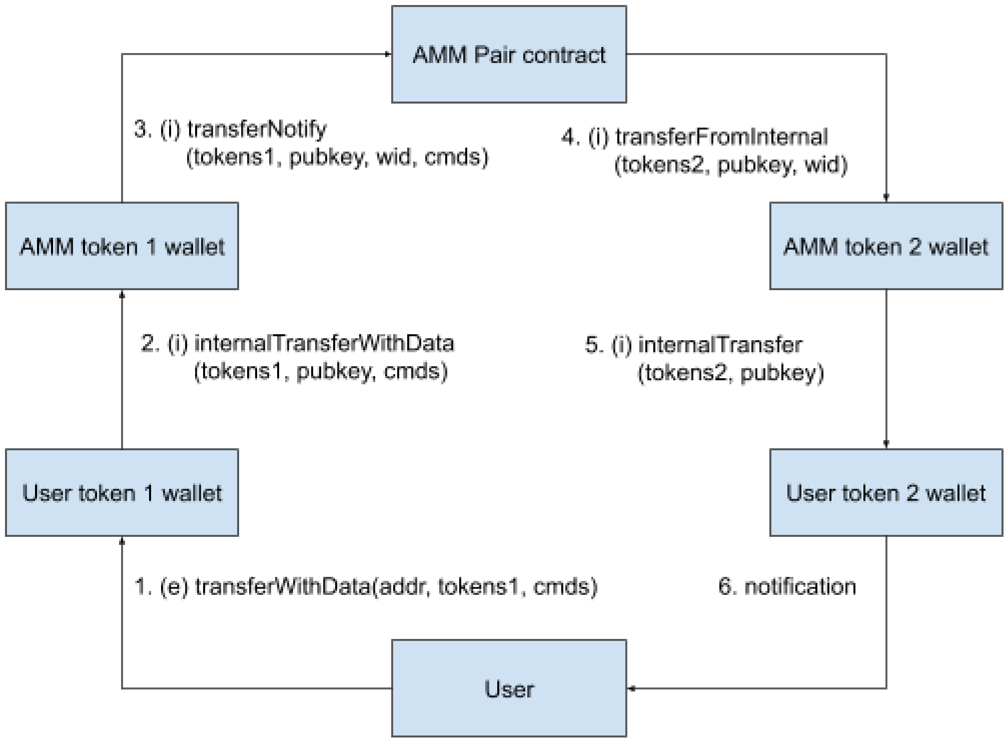
Commands are sent with a message in a sequential buffer with parameters. Special flags indicate about including specific parameters.

1. **CMD_EXCHANGE** - Exchange user’s tokens on opposite tokens. If successful - tokens are saved on the user's balance. Arguments:
    a. Minimal value (required) - minimal amount of opposite tokens that the user wants to get.
    b. Direction (required) - Direction of exchange, can be 1 to 2 or 2 to 1.
    c. Value (optional) - amount of user’s tokens for exchange. If not set, DEX will use the whole balance.

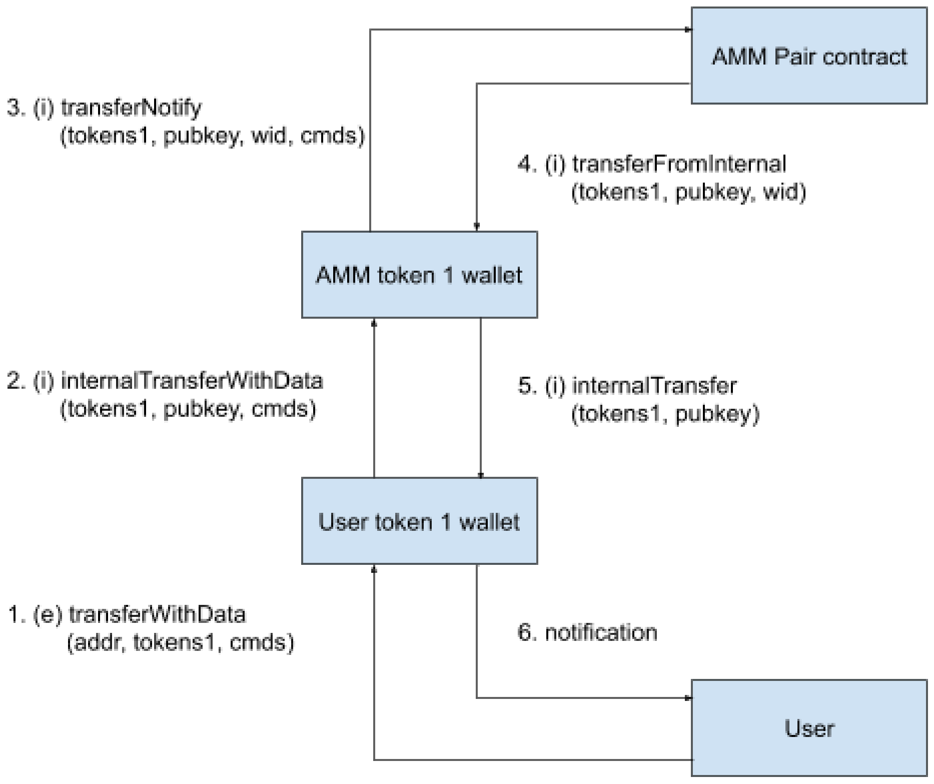
2. **CMD_LIQUIDITY_MINT** - mint liquidity tokens using the user's balances and deposit them in the liquidity pool in user's balance. Arguments:
    a. Value (optional) - maximal amount of liquidity tokens to mint. If not set, DEX will use the whole balance.
3. **CMD_LIQUIDITY_BURN** - burn liquidity tokens. The user gets a pair of tokens on his balances. Arguments:
    a. Value (optional) - amount of liquidity tokens to burn. If not set, DEX will use the whole balance of liquidity tokens.
4. **CMD_SEND_TOKENS** - send a token to the user. The user can send only one specific token at a time. Arguments:
    a. Token (required) - specific token will be used for sending.
    b. Value (optional) - amount of tokens to send. If not set, DEX will send the whole balance.
    c. Pubkey (optional) - public key and workchain id as destination of sending. If not set, DEX will sender's public key.

Tokens exchange

To exchange tokens the user need to send following command:
$CMD\_EXCHANGE + CMD\_SEND\_TOKENS(token\_destination) + CMD\_SEND\_TOKENS(token\_source)$
In case of success the opposite tokens will be sent to the user's wallet (Pic. 2). In case of failure, the user's tokens will come back (Pic. 3).


AMM Pair contract
3. (i) transferNotify (tokens1, pubkey, wid, cmds)
4. (i) transferFromInternal (tokens2, pubkey, wid)
AMM token 1 wallet
AMM token 2 wallet
2. (i) internalTransferWithData (tokens1, pubkey, cmds)
5. (i) internalTransfer (tokens2, pubkey)
User token 1 wallet
User token 2 wallet
1. (e) transferWithData(addr, tokens1, cmds)
6. notification
User

Pic. 2. Flow of success exchange


AMM Pair contract
3. (i) transferNotify (tokens1, pubkey, wid, cmds)
4. (i) transferFromInternal (tokens1, pubkey, wid)
AMM token 1 wallet
2. (i) internalTransferWithData (tokens1, pubkey, cmds)
5. (i) internalTransfer (tokens1, pubkey)
User token 1 wallet
1. (e) transferWithData (addr, tokens1, cmds)
6. notification
User

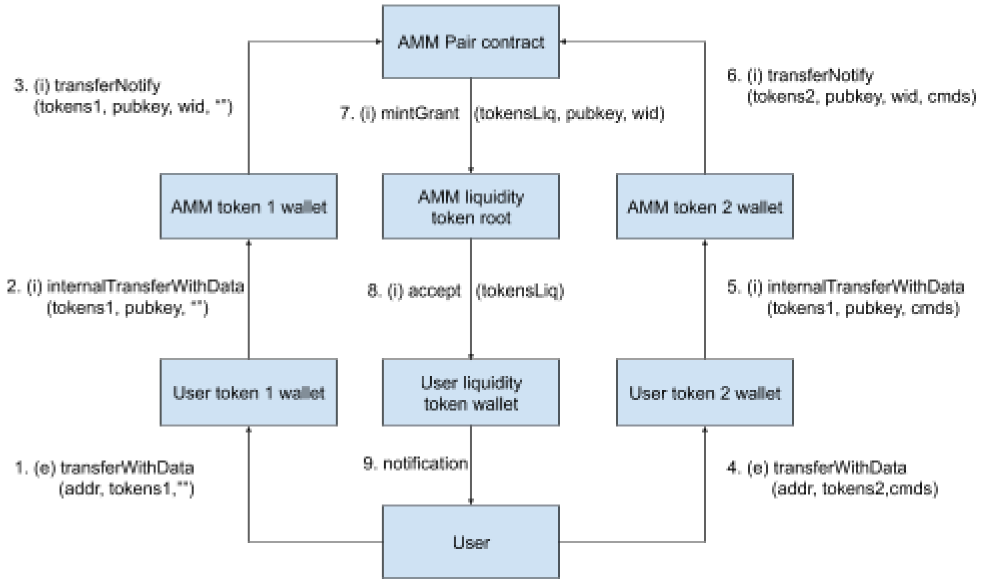
Pic. 3. Flow of failure exchange

Minting liquidity tokens

To mint liquidity tokens user has to pass two steps:

1. Deposit first tokens without commands.
2. Deposit second tokens with commands:

$$CMD\_LIQUIDITY\_MINT + CMD\_SEND\_TOKENS(token\_liquidity)$$


AMM Pair contract
3. (i) transferNotify (tokens1, pubkey, wid, "")
7. (i) mintGrant (tokensLiq, pubkey, wid)
6. (i) transferNotify (tokens2, pubkey, wid, cmds)
AMM token 1 wallet
AMM liquidity token root
AMM token 2 wallet
2. (i) internalTransferWithData (tokens1, pubkey, "")
8. (i) accept (tokensLiq)
5. (i) internalTransferWithData (tokens1, pubkey, cmds)
User token 1 wallet
User liquidity token wallet
User token 2 wallet
1. (e) transferWithData (addr, tokens1,"")
9. notification
4. (e) transferWithData (addr, tokens2,cmds)
User

Pic. 4. Minting liquidity tokens.

Calculation of liquidity tokens that user will get occurs using formula:

$$Tminted1 = (T1\_deposited / T1\_total) * Tliq\_total$$

$$Tminted2 = (T2\_deposited / T2\_total) * Tliq\_total$$

$$Tminted = min(Tminted1, Tminted2)$$

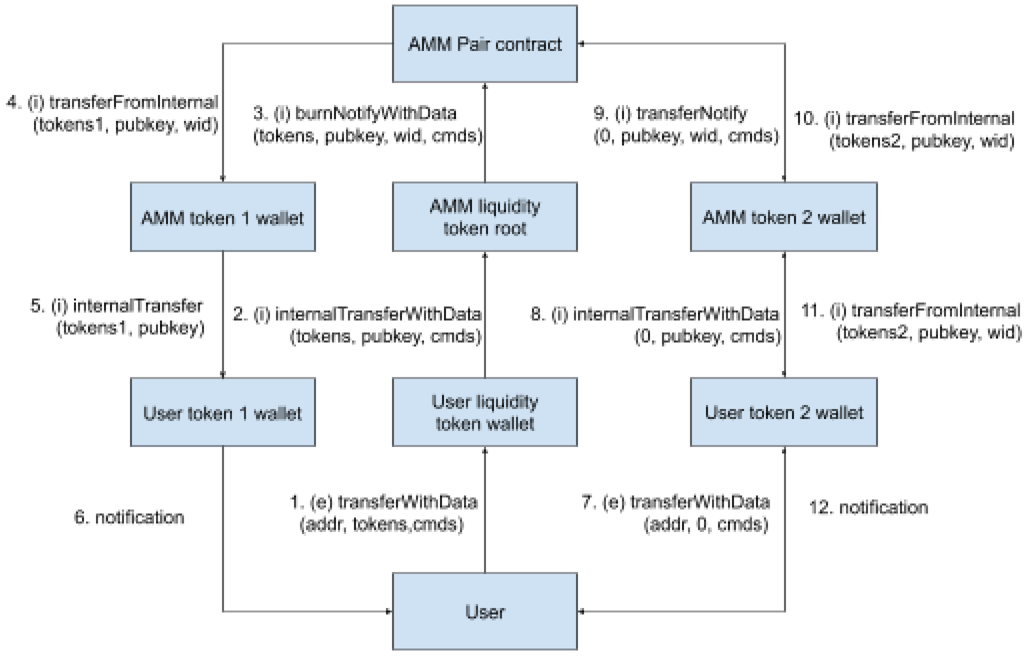
In case of first minging liquidity tokens:

$$Tminted = \sqrt{}(T1\_deposited * T2\_deposited)$$

Withdrawing liquidity tokens

To withdraw liquidity tokens user has to pass two steps:

1. Deposit liquidity tokens with commands:
$$CMD\_LIQUIDITY\_BURN + CMD\_SEND\_TOKENS(token\_1)$$
2. Withdraw second tokens using command: $CMD\_SEND\_TOKENS(token\_2)$.


AMM Pair contract
4. (i) transferFromInternal (tokens1, pubkey, wid)
3. (i) burnNotifyWithData (tokens, pubkey, wid, cmds)
9. (i) transferNotify (0, pubkey, wid, cmds)
10. (i) transferFromInternal (tokens2, pubkey, wid)
AMM token 1 wallet
AMM liquidity token root
AMM token 2 wallet
5. (i) internalTransfer (tokens1, pubkey)
2. (i) internalTransferWithData (tokens, pubkey, cmds)
8. (i) internalTransferWithData (0, pubkey, cmds)
11. (i) transferFromInternal (tokens2, pubkey, wid)
User token 1 wallet
User liquidity token wallet
User token 2 wallet
6. notification
1. (e) transferWithData (addr, tokens,cmds)
7. (e) transferWithData (addr, 0, cmds)
12. notification
User

Pic. 5. Withdrawing liquidity tokens.

Calculation of tokens that user will get occurs using formula:

$$T1 = Tliq * T1\_total / Tliq\_total$$
$$T2 = Tliq * T2\_total / Tliq\_total$$

Rebalancing

To mint liquidity tokens in the most effective way with minimal remainder there is the rebalancing mechanism. For that user has to have both tokens and bring them to the close required ratio. It can’t be without the remainder of tokens because of slippage.

To do that, user has to use command:

$$CMD\_EXCHANGE(target\_direction,\ target\_value)$$

Also the user can join this command with commands $CMD\_LIQUIDITY\_MINT\ +\ CMD\_SEND\_TOKENS(token\_liquidity)$ to mint and withdraw liquidity tokens just after rebalancing.

Impermanent loss

Impermanent loss occurs as a result of changing exchange ratio relatively when liquidity tokens was minted. It is named “Impermanent” because when the exchange ratio returns to its initial value then “loss” disappears. However, there is a possibility that the exchange ratio will not return to initial value. In this case loss must be compensated by exchange commission.

The danger arises in high volatility periods. In this case the arbitration mechanism returns the exchange ratio to the market ratio. So It makes no sense to defend yourself against this.

But anyway providers of liquidity have possibilities to wait out the volatility. To do that he must send the command **CMD_LIQUIDITY_BURN**, which burns liquidity tokens and enrolls pair tokens on his balance. After a while, he can rebalance pair tokens if needed and mint liquidity tokens again using command **CMD_LIQUIDITY_MINT**.

This approach to reduce "Impermanent loss" is practical because of low cost network fee (about 20k-30k at this moment). If many providers of liquidity will use this approach then the exchange ratio will stabilize faster because arbitration will work in conditions with low liquidity.

Also there are opportunities for users to delegate assets in trust management that will manage assets in an effective way and block assets in high volatility periods. Or it can be smart contracts which ensure safe funds and target profit.

Front-running

Front-running is possible on public blockchains such Ethereum or FreeTON for profit making. As a rule, a front-run message is created when a user's message is detected with the exchange command in the blockchain. The attacker must execute his commands before the user's message and affect the exchange ratio for attacker benefit. And the next attacker can make arbitration transactions from current or other exchanges.

There are differences between front running in Ethereum and FreeTON because in Ethereum you can increase the cost of gas to get to the closest block. In FreeTON there is no such mechanism but the attacker can use specific ordering of processing messages in software of validators.

For internal messages:

User’s wallets distribute across different shards so to deliver a user's message to exchange needs to pass at least 2 blocks. The attacker can create a wallet in the same shard as exchange and execute commands in 1 block before the user's message. If a user's wallet is placed in the same shard as an exchange so the attacker can use front-running with an external message.

For external messages:

All incoming messages are laid in map data type with ordering by hash of shard and hash of message. Collators take messages from the pool in ascending order by hash of message. That is the order in which they occur the block. So the attacker can create a message with a lower value of the message hash for executing before the user.

Nonetheless, there are several factors that make an attack difficult:

1. Asynchronous approach in message delivering makes impossible “flash loans” attacks .
2. So attacks in one transaction are impossible. Even though a transaction is splitted in several it has probabilistic nature and decreases profit expectation.
3. Protocol presented here has **required** the parameter minimal value of the opposite token that he wants to get. If the user sets the correct slippage value then the attack becomes unprofitable. If the user sets a high value of slippage so he must be informed about risks.

List of trading pairs

We offer to use a separate smart contract for accessing a list of trading pairs. It has a public interface and provides required information.

Mechanism of appending and deleting of trading pairs can be controlled in centralised (through public key of owner) or decentralised ways (using governance).

The UI of the exchange dapp must have the possibility to download information about trading pairs by user provided address.

Example of implementation is CharonPairList.sol, the test is test_pair_list.sh.

Governance

For managing a list of trusted trading pairs and changing commissions of exchange can be used governance smart contract. For this **AMM Pair contract** has an interface for changing commission within safe range.

Proposals for TIP-3

We propose to expand the interface of TIP-3 tokens to have ability integration with automatic smart contract systems such as exchanges, online shops, farming-services and etc.

RootTokenContract

__attribute__((internal, noaccept)) void mintGrant(TokensType tokens, uint256 pubkey, int8 workchain_id) = 30;
Method increases emission and sends *tokens* to the user. Address of the user wallet is computed using *pubkey* and *workchain_id*. Can be called only from another contract. Accept messages only from *root_public_key*. *root_public_key* can be a public key or hash part of address.

__attribute__((internal, noaccept)) void internalTransferWithData(TokensType tokens, uint256 pubkey, sequence<uint_t<8>> data) = 31;
Method burns *tokens* and notifies other smart-contract with *data*. *Pubkey* is used to ensure correct access from TIP-3 wallet.

__attribute__((internal, noaccept)) void setNotifyAddress(lazy<MsgAddressInt> addr) = 32;
Method sets *address* to internal memory, which will be used for sending notifications about minting or burning tokens. Contract must implement the ITONTokenNotify interface.

TONTokenWalllet

__attribute__((external, noaccept, dyn_chain_parse)) void transferWithData(lazy<MsgAddressInt> dest, TokensType tokens, WalletGramsType grams, sequence<uint_t<8>> data) = 30;
Method sends *tokens* to address *dest*. Can be called by external message signed *wallet_public_key*. Contract invokes *internalTransferWithData* with *data.*

`__attribute__((internal, noaccept))`
`void internalTransferWithData(TokensType tokens, uint256 pubkey, sequence<uint_t<8>> data) = 31;`

Method accept *tokens* and notify with *data* the other contract about that. Pubkey is used to ensure correct access from TIP-3 wallet.

`// send tokens from internal message`
`__attribute__((internal, noaccept))`
`void transferFromInternal(TokensType tokens, uint256 pubkey, int8 workchain_id) = 32;`

Method sends *tokens* to user using *pubkey* and *workchain_id* to compute TIP-3 wallet address. Can be invoked by internal message. Accept messages only from *wallet_public_key*. *wallet_public_key* can be a public key or hash part of address.

`__attribute__((internal, noaccept))`
`void setNotifyAddress(lazy<MsgAddressInt> addr) = 33;`

Method sets *address* to internal memory, which will be used for sending notifications about transfers in *internalTransferWithData*.

To use protocol for exchanging the pair “ token / TONCrystal” we suggest to create the TIP-3 wallet for TON Crystal by community.

Also we want to notice that interface with approve/internalTransferFrom is not useful for DEX due to increasing required message up 2 and absence of exact mechanism success transferring and time locking.

Interfaces for Exchange

AMM Pair Contract

constructor (address token1, address token2, address tokenLiqRoot, address developerAddress, address governanceAddress) public;
Token1 and *token2* addresses for TIP-3 exchange's wallets. *tokenLiqRoot* is a TIP-3 root contract for liquidity tokens. And address of developer and governance contracts.

function transferNotifyWithData(uint128 tokens, uint256 pubkey, int8 workchain_id, bytes data) external override functionID(100);
Method is invoked by exchange's wallets when refilled. Accept messages only from exchange's wallet *token1* and *token2*.

function burnNotifyWithData(uint128 tokens, uint256 pubkey, int8 workchain_id, bytes data) external override functionID(101);
Method is invoked by TIP-3 root contract of liquidity token after burning tokens. Accept messages only from the exchange's TIP-3 root contract.

Getters:

function getTokensAddress() public view returns (address token1Address, address token2Address, address tokenLiqRoot);
Return addresses of tokens.

function getReserves() public view returns (uint128 token1Reserve, uint128 token2Reserve, uint128 tokenLiqTotal);
Return current reserves.

function getFees() public view returns (uint128 exchangeFee, uint128 developerFee);
Return current exchange fees.

function getExchangeRate(uint128 valueToSwap, uint8 direction) public view returns (uint128 dstValue, uint128 serviceFee, uint128 developerFee, uint128 token1Reserve, uint128 token2Reserve);
Performs the calculation of exchange of one token to another. Returns the value of the exchanged token, the amount of commissions and current reserves.

function getBalance(uint256 pubkey) public view returns (uint128 token1Balance, uint128 token2Balance, uint128 tokenLiqBalance);
Returns the current balance of the user using "pubkey".

Governance interface:

function setExchangeFee(uint128 value) external governanceOnly;
Sets the exchange commission. Accepts messages only from the *governanceAddress*.

Developer interface:

function setDeveloperFee(uint128 value) external developerOnly;
Sets the developer commission. Accepts messages only from *developerAddress*.

function withdrawDeveloperFeeToken1(uint256 pubkey, int8 workchain_id, uint128 tokens) external developerOnly;
Withdraw token1 to the specified wallet from the developer's commission. Accepts messages only from *developerAddress*.

function withdrawDeveloperFeeToken2(uint256 pubkey, int8 workchain_id, uint128 tokens) external developerOnly;
Withdraw token2 to the specified wallet from the developer's commission. Accepts messages only from *developerAddress*.

function getDeveloperBalance() public view returns (uint128 token1, uint128 token2);
Returns the amount of accumulated developer fees.

AMM Pair List

constructor (bytes name, bytes description, bytes logoURI, bool ownerEnabled, address governanceAddress) public;
Init smart-contract. Flag *ownerEnabled* indicates that it's possible to manage the contract by owner.

Owner interface:

function addTokenOwner(address tokenAddress, bytes symbol, bytes name, bytes logoURI) external;
Add token in list. Accept messages only from *public_key* if *ownerEnabled=true*.

function delTokenOwner(address tokenAddress) external;
Remove token from list. Accept messages only from *public_key* if *ownerEnabled=true*.

function addPairOwner(address pairAddress, address token1Address, address token2Address, address tokenLiqAddress) external;
Add trading pair tokens in list. Accept messages only from *public_key* if *ownerEnabled=true*.

function delPairOwner(address pairAddress) external;
Remove trading pair tokens from list. Accept messages only from public_key if ownerEnabled=true.

Governance interface:

function addTokenGov(address tokenAddress, bytes symbol, bytes name, bytes logoURI) external;
Add token in list. Accept internal messages only from *governanceAddress*.

function delTokenGov(address tokenAddress) external;

Remove token from list. Accept internal messages only from *governanceAddress*.

function addPairGov(address pairAddress, address token1Address, address token2Address, address tokenLiqAddress) external;
Add trading pair tokens in list. Accept internal messages only from *governanceAddress*.

function delPairGov(address pairAddress) external;
Remove trading pair tokens from list. Accept internal messages only from governanceAddress.

Getters:

function getDescription() public view returns (bytes name, bytes description, bytes logoURI, bool ownerEnabled, address governanceAddress);
Return name and description of list.

function getTokenList() public view returns (address[] tokenAddress, bytes[] symbol, bytes[] name, bytes[] logoURI);
Return list of tokens.

function getPairList() public view returns (address[] pairAddress, address[] token1Address, address[] token2Address, address[] tokenLiqAddress);
Return list of trading pairs.