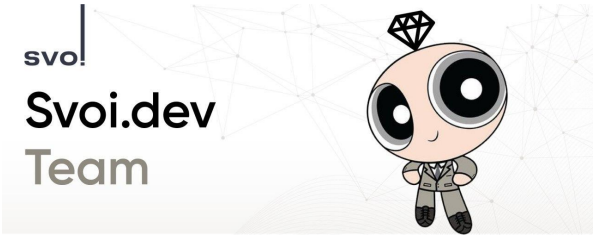


Russian version below.

PRESENTATION

Hello everyone!
We are SVOI.dev team and in this project we present the results of our long patient work,
submission to the contest:
#16 FreeTon DEX Implementation Stage 1 Contest.
The purpose of this work was the development and implementation of a fast exchange (Swap) in the FreeTON blockchain and infrastructure based on the “liquidity pool” approach.

Link to the project website: https://swap.tonswap.com
GitHub:
https://github.com/SVOIcom/tonswap-SC
https://github.com/SVOIcom/ton-testing-suite
https://github.com/SVOIcom/tonswap-frontend

Wallet: 0:0439186aa0147661ebaf2b32ecc76bac172fcdaa24c7df7c9cb03cc816e435e6

Contact information

Telegram (corporate account): https://t.me/svoidev
team’s website: https://SVOI.dev

Basic economic model and description of cash flow in the TonSwap system

As a result of the implementation of the TonSwap project presented by our team, an architecture for the implementation of a fast exchange (Swap) was created in the FreeTON blockchain and an infrastructure based on the general concept:

Automated Market Maker (AMM) using constant product formula.

Automated market makers are the backbone of the DeFi space. They enable just about anyone to create markets easily and efficiently. While they have their limitations when compared to order exchanges, the overall innovation they bring to cryptocurrency is invaluable.

AMM exchanges overcome the concept of an order book. Instead of setting prices by demand and supply, an AMM pools liquidity together and sets prices by way of a deterministic pricing formula.

We believe that this particular solution in terms of the DEX architecture is the most reasonable, profitable and effective for use in the TON network.

Let’s see from the economic perspective how such a model can incentivize users to use it for their daily exchange operations and at the same time benefit on the growth of the network in general and exchange in particular.

The basic motivation is obvious, people want to transfer their assets in blockchain depending on their needs and purposes, they want to exchange assets and in best case scenario earn yield rewards for providing liquidity.

On the other hand a good DEX implementation should ensure the stability of the exchange process, gain some dividends for active users and liquidity providers and provide some governance mechanism for future changes and improvements.

AMM (for ordinary users) can be thought of as a robot that is always ready to offer a price between two assets.

With the help of AMM, you can safely trade, as well as become the host of pairs / pools yourself, which allows almost anyone to become a market maker on the exchange and receive a commission for providing liquidity.

In simple terms, the TonSwap functioning model looks the next way:

1. liquidity providers place a pair of tokens - in existing liquidity pools or creating new ones;

- if a new pair is created, the primary liquidity provider sets the exchange rate of one token from the pair to another;
- in the case when the liquidity provider contributes tokens to an already existing pair (formed pool of liquidity) - tokens must be deposited into the pool in accordance with the exchange rate between the tokens of the pair that exists at the time the tokens are deposited into the pool;

2. when placing liquidity (a pair of tokens), suppliers receive a virtual “token” of the pair (Defined as the ratio `user-contributed liquidity` /` total liquidity`) - in confirmation of the share of participation in the liquidity pool, you can view this data by calling the pair's contract. In the future, it will be possible to replace it with the release of real tokens, which will go to the user's token wallet
3. traders / buyers, through the created Swap, can exchange their tokens for other tokens within the existing pairs at the rate calculated based on token amount in the swap pair/liquidity pool at the time of the operation;
4. When performing an exchange, a trader pays a commission, which distributes between the liquidity providers of the pool in which the exchange took place.
   The commission is distributed in accordance with the shares of the liquidity providers.
   The awards are determined by protocol. Currently, the commission is 0.3%, following the example of Uniswap v2, which may later change. Other platforms or forks may charge lower fees in order to attract more liquidity providers to their pool.
5. when changing one token to another in the established pool, their ratio within the pool changes depending on the volume of entered and contributed tokens;

* the implementation of an exchange transaction is possible in the case when the buyer of tokens in the wallet has a sufficient volume of another type of tokens of the pair for exchange, as well as to cover the commission paid to liquidity providers;

6. For operations to not have a large volume of influence on one another - for the exchange the most effective conditions of functioning, when:

- there are a large number of liquidity providers and buyers / traders
- the exchange operations are carried out in small amounts - the developed pool of liquidity of the pair.

But these are conditions that cannot be directly influenced.

7. It is important to describe one more of the actions carried out within the Swap Pair - withdrawal by the liquidity provider of the embedded tokens:
   if at a certain moment the liquidity provider decides to withdraw his tokens from the pool, then at the moment he owns an amount of tokens not equal to initial one, but equal to its share of the total volume.

For example:

The liquidity provider N initially invests in the pool a pair of tokens - 1 WETH and 1 WBTC - based on the established rate of 1: 1.

at the time of entering the pool, 9 WETH and 9 WBTC have already been invested, those his share in the pool is 10%.

Suppose, for some time, there is a numerous exchange of some tokens for others, as a result of which we get the following ratio of the pair's tokens:

6 WETH and 20 WBTC.

If, during the time period of this established ratio, the liquidity provider N decides to withdraw his tokens from the pool, then he receives them in the amount:

0.6 WETH and 2 WBTC.

On the Efficiency of Automated Market Makers

In AMM models, the price is determined by deterministic formulas based on the supply and demand of the assets. As a result, these models rely on the arbitrageurs to level the price with the other markets. In other words, when there is a price difference, the arbitrageurs start buying/selling assets from the smart contract, which changes the supply and demand of an asset and levels the price. However, this results in inefficiency for liquidity providers as they endure losses (or opportunity costs relative to the true market price that the arbitrageurs utilize).

All of the above can be presented in the form of a basic formula:

We have a market which is for trading coins of type α for coins of type β (and vice versa). This market has reserves Rα > 0 and Rβ > 0, constant product k = RαRβ, and percentage fee (1 − γ). A transaction in this market, trading ∆β > 0 coins β for ∆α > 0 coins α, must satisfy

$$(R_\alpha - \Delta_\alpha)(R_\beta + \gamma\Delta_\beta) = k.$$

After each transaction, the reserves are updated in the following way: Rα "→ Rα − ∆α, Rβ "→ Rβ + ∆β, and k "→ (Rα − ∆α)(Rβ + ∆β). We will always require that Rα,Rβ > 0, such that any trade that results in a nonpositive reserve is never fulfilled (equivalently, we say that such a trade has infinite cost).

When the fee is zero (i.e., γ = 1), any trade ∆β to ∆α must change the reserves in such a way that the product RαRβ remains equal to the constant k.

Description of the system of interaction between smart contracts

The TIP-3 token exchange system based on liquidity pairs implies the following smart contracts:

1. TIP-3 token contracts - root contracts and wallet contracts;
2. Exchange contracts on liquidity pairs - root contract and exchange pairs contracts;
3. TON wallet contracts.

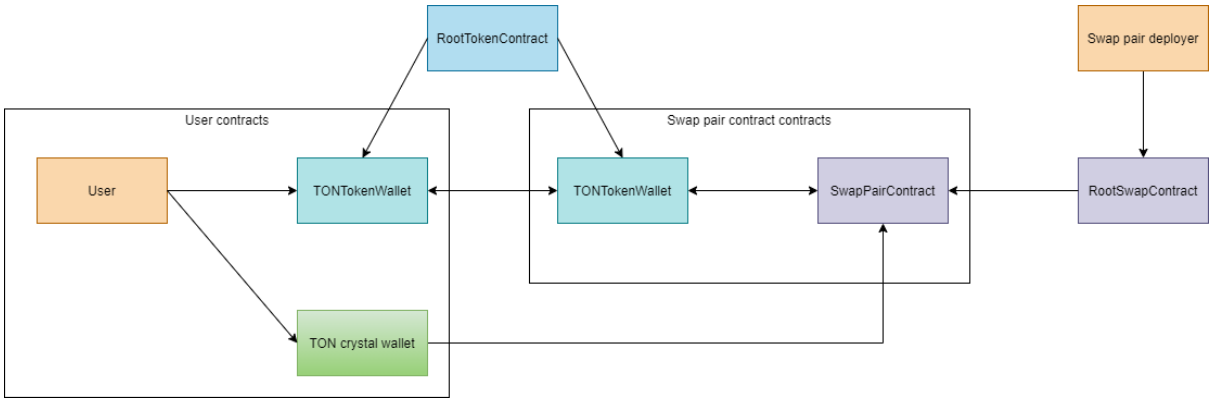


Fig.1 - general scheme of interaction of smart contracts

Description of the interaction of the root pair - pair

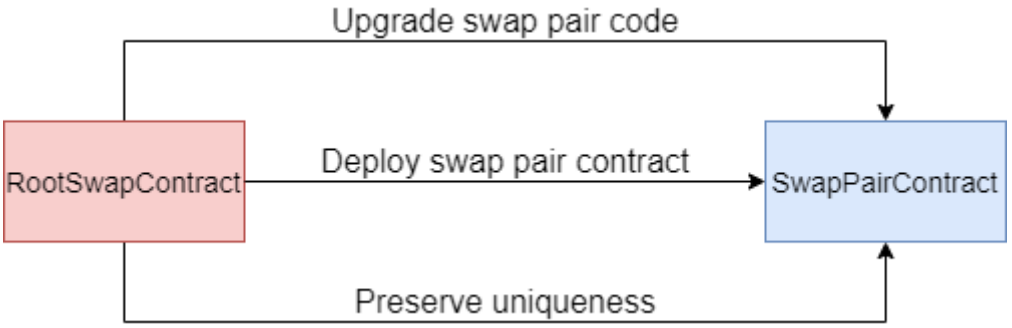


Fig. 2 - the scheme of interaction between the root swap pair contract and the swap-pair contract

The swap pair contract and swap pair root contract interact as follows:

The root contract performs a pair deployment with the allocation of some initial balance of TONs, which is spent on deploying TIP-3 wallets and some initial configuration: obtaining wallet addresses from root token contracts and setting an address for callbacks. The root contract is also used to update the swap pair code.

Description of the interaction of the pair - token

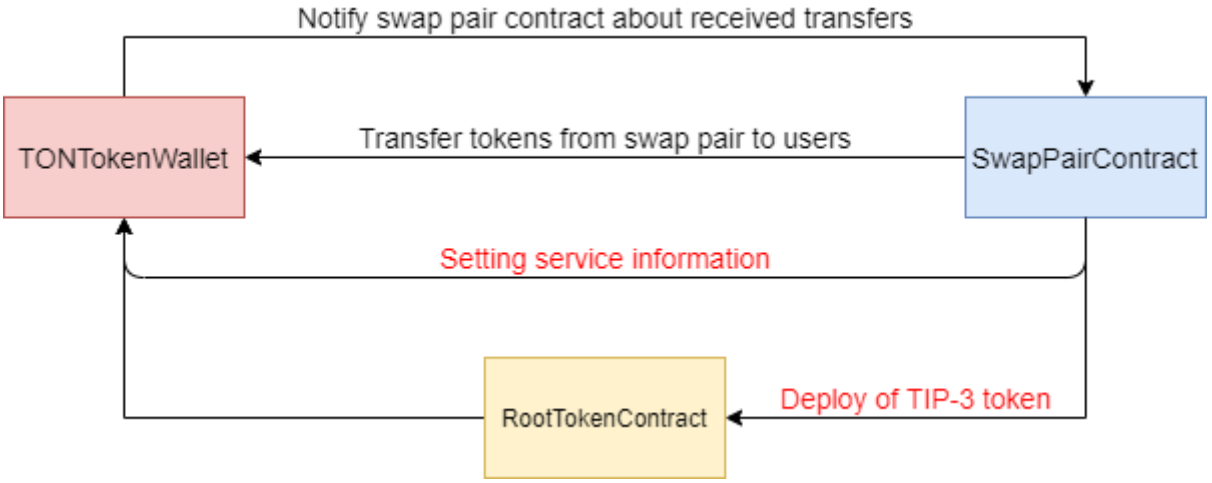


Fig. 3 - a description of the interaction between the TIP-3 wallet and the swap pair. The interactions that consume the initially allocated balance of the swap pair are marked in red

The interaction of the swap-pair contract and the token consists in transferring tokens to users and obtaining information about the transferred tokens.

When transferring tokens to a TIP-3 wallet belonging to the swap pair, a callback for receiving the transfer is called. The transfer of tokens to users is initiated in the swap pair by calling the appropriate method, after which the TIP-3 wallet is called from the swap pair contract.

Description of interaction TON wallet - pair

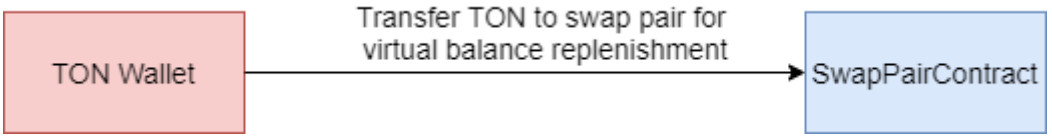


Fig. 4 - interaction between TON wallet and swap pair

The interaction of the TON wallet contract and the swap pair is to replenish the user's internal balance. The swap-pair supports the processing of external messages (since they can be used when interacting with the swap-pair using special sites or debots), therefore, an internal user balance is required, which will limit the interaction between the user and the pair's contract so that the contract does not end up in an inactive state.

The mechanism for charging a fee for performing certain actions is described in the corresponding section of the description of the swap pair.

When transferring, you must attach a payload containing the user's public key in the unit256 format, if the public key could not be obtained, only a part of the received TONs is sent back.

The required payload can be formed as follows:

```
TvmBuilder builder;
builder.store(uint256(userPubkey));
TvmCell payload = builder.toCell();
```

Description of the user-couple interaction

The scenario of interaction between the user and the swap pair looks like this (if there is an existing pair):

1. Replenishment of your virtual balance of TONs in a swap pair to perform operations to provide liquidity, withdraw liquidity from the pool and exchange tokens. This stage is performed by transferring TONs to a swap-pair contract with a payload containing the user's public key;
2. Replenishment of your virtual balance of tokens for swap by transferring tokens from the TIP-3 wallet to the swap-pair wallet. At the moment, it is necessary to perform a transfer using the **`transferWithNotify`** function so that the swap-pair contract is notified of the transfer;
3. Execution of available operations with tokens: deposit / withdrawal of liquidity, exchange (swap);
4. Withdrawing tokens through a request to a swap pair.

RootSwapPairContract

A brief description of the smart contract

This smart contract is designed to create swap pairs, store information about existing swap pairs, and also to update swap pairs when a new version is released.

Constructor and static variables

The contract contains the following static variables:

1. uint256 ownerPubkey - the public key of the owner of the swap-pair root contract.

The constructor takes the following parameters as input:

1. TvmCell spCode - TvmCell containing the code of the swap-pair contract;
2. uint32 spCodeVersion - version of the swap pair contract code;
3. uint256 minMsgValue - the minimum amount of TONs attached to the function call for deploying the swap pair;
4. uint256 contractSP - the amount of the contract service fee, which is deducted from the cost of the incoming message.

Functions

deploySwapPair - this function is used to deploy a swap pair of a contract. The end result does not depend on the order of the input parameters - to calculate the address of the future contract of the pair, the static field uniqueId is used as a static variable, calculated as:

$$uniqueId = tokenRootContract.value \oplus tokenRootContract.value;$$

This ensures the uniqueness of the swap pair within a single swap root contract.

The input takes the following parameters:

1. address tokenRootContract1 - address of the token's root contract;
2. address tokenRootContract2 - address of the token's root contract.

The function returns:

1. address cA - the address of the swap-pair defaulted.

getPairInfo - this function is used to get information about existing swap pairs.

Function arguments:

1. address tokenRootContract1 - address of the root contract of token 1;
2. address tokenRootContract2 - address of the root contract of token 2.

The function returns:

struct SwapPairInfo:

1. address rootContract - the address of the root contract;
2. address tokenRoot1 - address of the root contract of token 1;
3. address tokenRoot2 - address of the root contract of token 2;
4. address tokenWallet1 - the address of the wallet of token 1 (equal to the zero address when requesting information from the root contract);
5. address tokenWallet2 - the address of the wallet of token 2 (equal to the zero address when requesting information from the root contract);
6. uint256 deployerPubkey - the public key of the one who initiated the deployment of the swap pair;
7. uint256 deployTimestamp - deployment time of the swap pair;
8. address swapPairAddress - swap-pair address;
9. uint256 uniqueId - unique number of the pair;
10. uint32 swapPairCodeVersion - version of the pair code.

getServiceInformation - getting more service information about the root contract.

Function arguments: no arguments
The function returns:
struct ServiceInfo:
1. uint256 ownerPubkey - the public key of the root contract deployer;
2. uint256 contractBalance - balance in TONs;
3. uint256 creationTimestamp - contract creation time;
4. uint32 codeVersion - swap-pair code version;
5. TvmCell swapPairCode - swap-pair code.

checkIfPairExists - function for checking the existence of a swap pair with the specified addresses of token root contracts.

Function arguments:
1. address tokenRootContract1 - address of the root contract of token 1;
2. address tokenRootContract2 - address of the root contract of token 2.

The function returns:
1. bool value0 - whether there is a swap pair.

setSwapPairCode - function for setting new swap pair code at the root contract level. When attempting to update a pair, the code stored in the pair will be used. The function can only be called by those who have deployed the original root swap pair contract.
Function arguments:
1. TvmCell code - new code for the swap pair smart contract;
2. uint32 codeVersion - smart contract code version.

upgradeSwapPair - function for updating the swap-pair contract by its unique number using the existing swap-pair code. Can only be triggered by those who have deployed the swap pair.

Function arguments:
1. uint256 uniqueID - unique number of the pair.

SwapPairContract

A brief description of the smart contract

This smart contract is designed to implement the logic of the exchange on liquidity pools and provide users with access to this functionality.

Features of the smart contract

As previously mentioned, a contract requires a payment for its execution in order for it to function. In order to use the functionality of liquidity pools, the user must first deposit TONs on his virtual balance of the swap pair. However, the question arises as to what size of the fee needs to be set. It was decided to implement a mechanism for balancing fees for using liquidity pools. It works as follows: the contract balance is compared before and after the performance of “heavy” functions, after which the cost of their performance is changed. The cost is increasing faster than it is decreasing. An example of functioning is shown in the graph below:

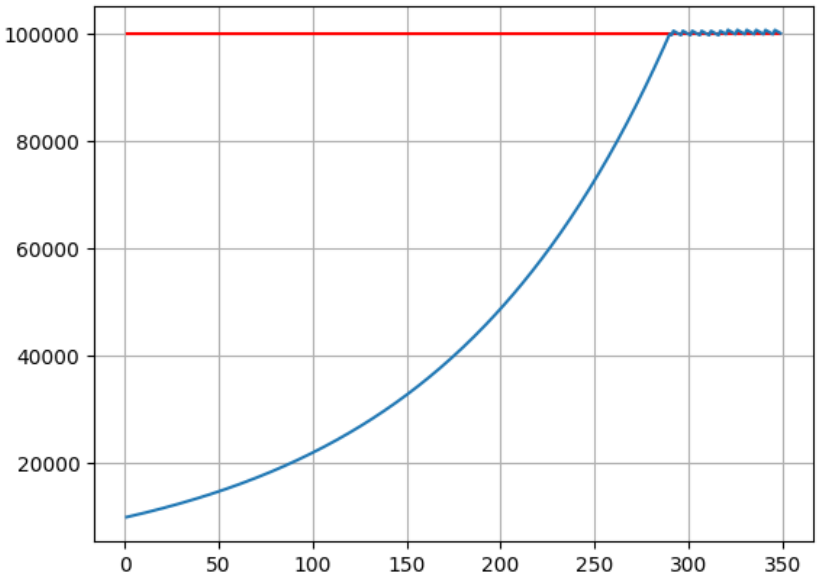


Fig. 5 - functioning of the mechanism for changing the cost of execution

As you can see, the cost quickly reduces to the average cost of executing the function, after which it begins to fluctuate within a small area around the real value of the cost of executing.

This mechanism allows not only to dynamically adjust the cost of performing functions, but also helps to respond to changes in the balance of the contract, maintaining it in working order.

Constructor and static variables

The contract contains the following static variables:

1. address token1 - address of the root contract of token 1;
2. address token2 - address of the root contract of token 2;
3. uint256 swapPairId - unique pair number;

The constructor takes the following parameters as input:

1. address rootContract - address of the root contract;
2. uint256 spd - public key of the contract deployment initiator.

Functions

_reinitialize - if the creation of the pair's wallets has not been completed, you need to call this function to start reinitializing the contract. Only the one who has deployed the swap pair can call the function. At least 2 Tons must be attached to the message.

Function arguments: no arguments

getCreationTimestamp - returns the creation time of the swap pair.

The function returns:
1. uint256 value0 - time of swap-pair creation.

getPairInfo - returns information about the swap pair.

The function returns:
struct SwapPairInfo:
1. address rootContract - address of the root contract;
2. address tokenRoot1 - address of the root contract of token 1;
3. address tokenRoot2 - address of the root contract of token 2;
4. address tokenWallet1 - wallet address of token 1;
5. address tokenWallet2 - address of token wallet 2;
6. uint256 deployerPubkey - the public key of the creator of the pair;
7. uint256 deployTimestamp - pair creation time.
8. address swapPairAddress - swap-pair address;
9. uint256 uniqueId - unique number of the swap pair;
10. uint32 swapPairCodeVersion - swap-pair code version.

getUserBalance is a function for getting the balance of user tokens that are not in the swap pair's liquidity pool. Either the passed parameter pubkey (for internal messages) or a signed message and parameter pubkey = 0 (for external messages) is required.

Function arguments:
1. uint256 pubkey is the user's public key.

The function returns:
struct UserBalanceInfo
1. address tokenRoot1 - address of the root contract of token 1;
2. address tokenRoot2 - address of the root contract of token 2;
3. uint128 tokenBalance1 - user's balance in token 1;
4. uint128 tokenBalance2 - user's balance in token 2.

getUserTONBalance - a function to get the balance of the user's TONs in a swap pair. Either the passed parameter pubkey (for internal messages) or a signed message and parameter pubkey = 0 (for external messages) is required.

Function arguments:
1. uint256 pubkey - user's public key;

The function returns:
1. uint256 balance - user balance in TONs.

getUserLiquidityPoolBalance is a function for getting the balance of the user's tokens in the swap pair's liquidity pool. Either the passed parameter pubkey (for internal messages) or a signed message and parameter pubkey = 0 (for external messages) is required.

Function arguments:

1. uint256 pubkey is the user's public key.

The function returns:

struct UserBalanceInfo

1. address tokenRoot1 - address of the root contract of token 1;
2. address tokenRoot2 - address of the root contract of token 2;
3. uint256 userLiquidityTokenBalance - the amount of tokens received as a result of adding liquidity to the pair (currently some conventional units);
4. uint256 liquidityTokensMinted - the total amount of issued tokens (currently - conventional units) of the liquidity pool;
5. uint128 lpToken1 - amount of token 1 in the liquidity pool;
6. uint128 lpToken2 - amount of token 2 in the liquidity pool.

The user's percentage of pool ownership is calculated using the following formula:

$$userLPpercent = userLiquidityTokenBalance/liquidityTokensMinted;$$

The amount of tokens in the liquidity pool owned by the user:

$$userLPToken1 = userLPpercent * lpToken1;$$

When calculating in smart contracts, it is recommended to use the math.muldiv function to avoid loss of precision in calculations.

getExchangeRate - getting the result of exchange of a specified amount of tokens.

Function arguments:

1. address swappableTokenRoot - selected token for swap;
2. uint128 swappableTokenAmount - the specified amount of tokens to exchange.

The function returns:

struct SwapInfo

1. uint128 swappableTokenAmount - the amount of tokens used for swap;
2. uint128 targetTokenAmount - the amount of tokens that can be obtained as a result of the swap;
3. uint128 fee - swap fee.

getProvidingLiquidityInfo - simulates adding tokens to the liquidity pool to determine the resulting amount of added tokens.

Function arguments:

1. uint128 maxFirstTokenAmount - the maximum amount of tokens 1 added to the liquidity pool;
2. uint128 masSecondTokenAmount - the maximum amount of tokens 2 contributed to the liquidity pool.

The function returns:

1. uint128 providedFirstTokenAmount - the amount of 1 tokens that would be contributed to the liquidity pair;
2. uint128 providedSecondTokenAmount - the amount of tokens 2 that would be contributed to the liquidity pair.

getWithdrawLiquidityInfo - simulation of withdrawal of tokens from the liquidity pool to determine the resulting amount of withdrawn tokens.

Function arguments:

1. uint128 minFirstTokenAmount - the minimum amount of tokens 1 to withdraw from the liquidity pool;
2. uint128 minSecondTokenAmount - the minimum amount of tokens 2 to withdraw from the liquidity pool;

The function returns:

1. uint128 - withdrawedFirstTokenAmount - the amount of tokens 1 that were withdrawn from the liquidity pool;
2. uint128 - withdrawedSecondTokenAmount - the amount of tokens 2 that have been withdrawn from the liquidity pool.

provideLiquidity - adding tokens to the swap pair liquidity pool.

Function arguments:

1. uint128 maxFirstTokenAmount - the maximum amount of tokens 1 added to the liquidity pool;
2. uint128 maxSecondTokenAmount - the maximum amount of tokens 2 added to the liquidity pool.

The function returns:

struct SwapInfo

1. uint128 providedFirstTokenAmount - the amount of tokens 1 contributed to the liquidity pair;
2. uint128 providedSecondTokenAmount - the amount of tokens 2 contributed to the liquidity pair.

withdrawLiquidity - withdrawal of tokens from the swap pair liquidity pool.

Function arguments:

1. uint256 liquidityTokensAmount - the amount of virtual LP-tokens to withdraw the proportional amount of tokens from the liquidity pool;

The function returns:

1. uint128 withdrawedFirstTokenAmount - the amount of tokens 1 that were withdrawn from the liquidity pool;
2. uint128 withdrawedSecondTokenAmount - the amount of tokens 2 that have been withdrawn from the liquidity pool.

swap is a function for exchanging a given number of selected amount of tokens for another token.

Function arguments:

1. uint128 swappableTokenRoot - address of the root contract of the selected token for swap;
2. uint128 swappableTokenAmount - the amount of tokens to exchange.

The function returns:

struct SwapInfo

1. uint128 swappableTokenAmount - the amount of tokens used for swap;
2. uint128 targetTokenAmount - the amount of tokens received as a result of the swap;
3. uint128 fee - swap fee deducted from received tokens.

withdrawTokens - withdrawal of a specified amount of tokens from a pair. Tokens that are not currently in the liquidity pool are available for withdrawal.

Function arguments:

1. address withdrawalTokenRoot - address of the root contract of the selected token;
2. address receiveTokenWallet - wallet for receiving tokens;
3. uint128 amount - the amount of tokens to withdraw.

getWalletAddressCallback is a callback function to get the address of the token wallet. Can only be triggered by the root contracts of the pair's tokens.

Function arguments:

1. address walletAddress - wallet address.

tokensReceivedCallback is a callback function for receiving information about token transfers. Can only be triggered by wallets.

Function arguments:

1. address token_wallet - token wallet address;
2. address token_root - address of the token's root contract;
3. uint128 amount - the amount of tokens received;
4. uint256 sender_public_key - public key of the sender of tokens, which will be used to add tokens to the user's account;
5. address sender_address - sender's address;
6. address sender_wallet - sender wallet address;
7. address original_gas_to - address where surplus gas will be sent;
8. uint128 updated_balance - wallet balance after receiving tokens;
9. TvmCell payload - attached additional information.

updateSwapPairCode - function for updating the swap-pair code. Can only be triggered by a root contract.

Function arguments:

1. TvmCell newCode - TvmCell containing the new code for the swap-pair contract;
2. uint32 newCodeVersion - new version of the swap-pair contract code.

Swap pair debot

General description

For the convenience of user interaction with the swap-pair contract, a smart contract was developed for debot. This smart contract greatly simplifies interaction with the swap pair contract, since it does not require the user to write any code. The only requirement when using a debot is the preliminary transfer of TONs and tokens to the swap-pair contract, since this debot does not support these operations.

Debot functionality

With this debot, you can perform the following operations:

1. Obtaining information about available tokens to be added to the liquidity pool or exchange;
2. Obtaining information about tokens in the liquidity pool;
3. Obtaining information about user's TON balance;
4. Obtaining information about execution cost of functions related to liquidity pools (providing liquidity, removing liquidity, swap, withdrawing tokens);
5. Adding tokens to the liquidity pool;
6. Withdrawing tokens from the liquidity pool;
7. Getting current exchange rate;
8. Token exchange;
9. Withdrawing tokens from a pair.

ПРЕДСТАВЛЕНИЕ

Всем привет!
Мы команда SVOI.dev и в представленном проекте - результаты нашего длительного кропотливого труда, осуществляемого в рамках контеста, проводимого FreeTON Community:
#16 FreeTon DEX Implementation Stage 1 Contest.
Целью данной работы стала разработка и имплементация быстрого обмена (Swap) в блокчейне FreeTON и инфраструктуры, базирующегося на основе подхода “пул ликвидности”.

Ссылка на сайт проекта: https://swap.tonswap.com
GitHub:
https://github.com/SVOIcom/tonswap-SC
https://github.com/SVOIcom/ton-testing-suite
https://github.com/SVOIcom/tonswap-frontend

Кошелек: 0:0439186aa0147661ebaf2b32ecc76bac172fcdaa24c7df7c9cb03cc816e435e6

Контактная информация

Telegram (корпоративный акк): https://t.me/svoidev
сайт команды: https://SVOI.dev

Базовая экономическая модель и описание денежного потока

В результате реализации нашей командой представленного проекта - TonSwap - была создана архитектура имплементации быстрого обмена (Swap) в блокчейне FreeTON и инфраструктура базирующиеся на основе общей концепции:
Automated Market Maker (AMM) с использованием формулы константного продукта.

Автоматизированные маркет-мейкеры являются основой пространства DeFi. Они позволяют практически любому легко и эффективно создавать рынки. Хотя у них есть свои ограничения по сравнению с биржами заказов, общие инновации, которые они привносят в криптовалюту, неоценимы.

Биржи AMM превосходят концепцию книги заказов. Вместо того, чтобы устанавливать цены на основе спроса и предложения, AMM объединяет ликвидность и устанавливает цены с помощью детерминированной формулы ценообразования.

Считаем, что именно данное решение в части DEX архитектуры - наиболее разумное, выгодное и эффективное для использования в сети TON.

Давайте посмотрим с экономической точки зрения, как такая модель может побудить пользователей использовать ее для своих повседневных операций обмена и в то же время выиграть от роста сети в целом и обмена в частности.

Основная мотивация очевидна: люди хотят переносить свои активы в блокчейне в зависимости от своих потребностей и целей, они хотят обменять актив и в лучшем случае получить доход за предоставление ликвидности.

С другой стороны, хорошая реализация DEX должна обеспечить стабильность процесса обмена, принести некоторые дивиденды для активных пользователей и поставщиков ликвидности и предоставить некоторый механизм управления для будущих изменений и улучшений.

AMM (для простых пользователей) можно представить как робота, который всегда готов предложить цену для обоих активов.

С помощью AMM можно безопасно торговать, а также самому становиться хостом пар / пулов, что позволяет практически любому стать маркет-мейкером на бирже и получать комиссию за предоставление ликвидности.

Простым языком модель функционирования TonSwap выглядит следующим образом:

1. поставщики ликвидности размещают пару токенов - в существующие пулы ликвидности или образовывая новые;

- в случае создания новой пары - первичный поставщик ликвидности задаёт курс обмена одного токена из пары на другой;
- в случае, когда поставщик ликвидности вносит токены в уже существующую пару (образованный пул ликвидности) - токены должны вносится в пул в соответствии с обменным курсом между токенами пары, существующим на момент внесения токенов в пул;

2. при размещении ликвидности (пары токенов) поставщики получают виртуальный “токен” пары (Определяется как отношение`вложенная пользователем ликвидность` / `общая ликвидность`) - в подтверждение доли участия в пуле ликвидности, просмотреть эти данные можно обратившись в контракт пары. В последующем можно будет заменить на выпуск реальных токенов, которые будут поступать на пользовательский токен кошелёк;

3. трейдеры / покупатели через созданный Swap могут обменять имеющиеся у них токены на другие токены в рамках существующих пар по курсу, рассчитанному на основании объема токенов в свап паре / пуле ликвидности на момент осуществления операции;

4. при осуществлении обмена трейдером уплачивается комиссия, сумма которой распределяется между поставщиками ликвидности пула, в котором произошёл обмен.

Распределение комиссии происходит в соответствии с долями поставщиков ликвидности.

Награды определяются протоколом. В настоящее время комиссия составляет 0,3%, по примеру Uniswap v2, в последующем размер комиссии можно изменить. Другие платформы или форки могут взимать меньшую плату, чтобы привлечь больше поставщиков ликвидности в свой пул.

5. при осуществлении обмена одного токена на другой в рамках определённого пула - изменяется их соотношение внутри пула в зависимости от объёма выведенных и внесённых токенов;

*осуществление транзакции обмена возможно в случае, когда у покупателя токенов в кошельке достаточный объём другого вида токенов пары для обмена, а также для покрытия комиссии, уплачиваемой поставщикам ликвидности;

6. чтобы операции обмена не имели большого влияния на стоимость одного токена по отношению к другому - для Swap наиболее благоприятные условия функционирования, когда:

- существует большое количество поставщиков ликвидности и покупателей / трейдеров
- проведение операций обмена происходит в суммах небольших - относительного сформированного пула ликвидности пары.

Но это условия, на которые прямо повлиять нет возможности.

7. Важно описать ещё одно из действий, осуществляемых в рамках Swap Pair - выведение поставщиком ликвидности вложенных токенов:

если в определённый момент поставщик ликвидности принимает решение вывести свои токены из пула, то ему предоставляется объём токенов не равный первоначальному, а равный его доле от общего объёма.

Например:

Поставщик ликвидности N первоначально вкладывает в пул пару токенов - 1 WETH и 1 WBTC - исходя из установленного курса 1:1.

на момент внесения в пул уже вложено 9 WETH и 9 WBTC, т.е. его доля в пуле равна 10%.

Допустим в течение какого-то времени происходит многочисленный обмен одних токенов на другие в результате чего получаем следующее соотношение токенов пары:

6 WETH и 20 WBTC.

Если в период времени данного установленного соотношения поставщик ликвидности N принимает решение вывести свои токены из пула, то он получает их в объёме:

0.6 WETH и 2 WBTC.

Об эффективности автоматизированных маркет-мейкеров

В моделях AMM цена определяется по детерминированным формулам, основанным на спросе и предложении активов. В результате эти модели полагаются на арбитражеров, чтобы уравнять цену с другими рынками. Другими словами, когда есть разница в цене, арбитражёры начинают покупать / продавать активы из смарт-контракта, который изменяет спрос и предложение актива и выравнивает цену. Однако это приводит к неэффективности поставщиков ликвидности, поскольку они несут убытки (или альтернативные издержки по сравнению с реальной рыночной ценой, которую используют арбитражёры).

Все вышеперечисленное можно представить в виде базовой формулы:

У нас есть рынок, на котором можно обменять монеты типа α на монеты типа β (и наоборот). Этот рынок имеет резервы Rα> 0 и Rβ> 0, постоянный продукт k = RαRβ и процентную плату (1 - γ). Сделка на этом рынке, торгующая ∆β> 0 монет β для ∆α> 0 монет α, должна удовлетворять

$$(R_\alpha - \Delta_\alpha)(R_\beta + \gamma\Delta_\beta) = k.$$

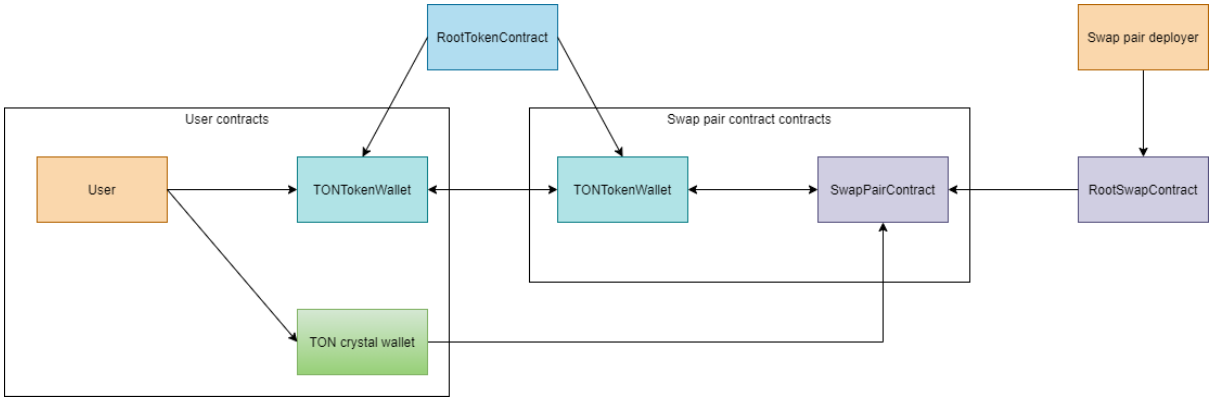
После каждой транзакции резервы обновляются следующим образом: Rα "→ Rα - ∆α, Rβ" → Rβ + ∆β и k "→ (Rα - ∆α) (Rβ + ∆β). Мы всегда будем требовать что Rα, Rβ> 0, так что любая сделка, которая приводит к неположительному резерву, никогда не выполняется (эквивалентно, мы говорим, что такая сделка имеет бесконечную стоимость).

Когда комиссия равна нулю (т. Е. Г = 1), любая сделка ∆β на ∆α должна изменять резервы таким образом, чтобы продукт RαRβ оставался равным константе k.

Описание системы взаимодействия между смарт-контрактами

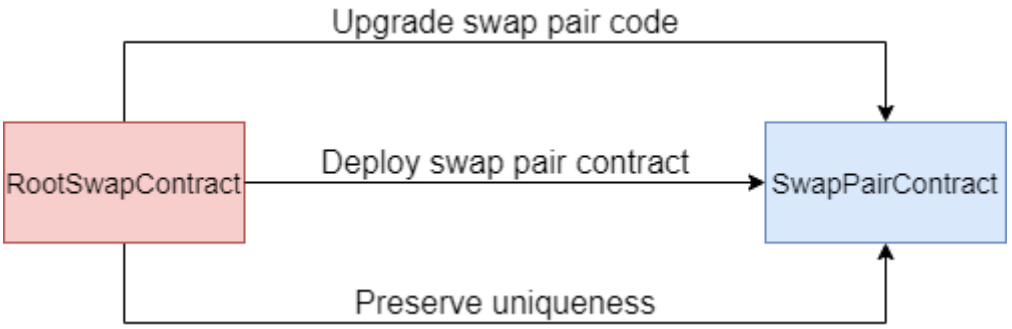
Система обмена TIP-3 токенами на основе пар ликвидности подразумевает наличие следующих смарт-контрактов:

1. Контракты TIP-3 токенов - рут-контракты (root contracts) и контракты-кошельков;
2. Контракты обмена на парах ликвидности - рут-контракт и контракты пар обмена;
3. Контракты TON-кошельков.



Изображение 1 - Схема взаимодействия смарт-контрактов

Описание взаимодействия рут пары - пара

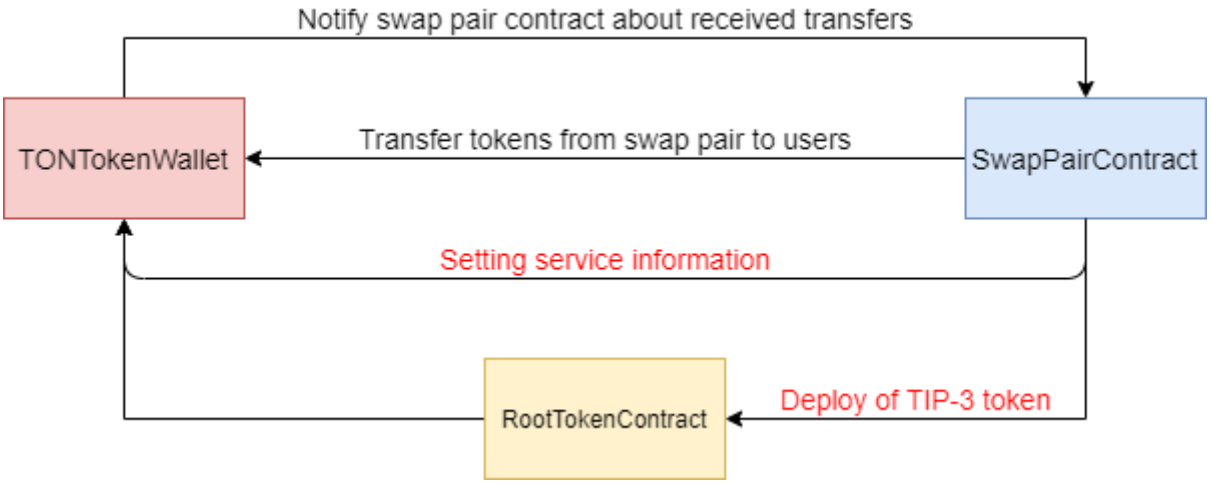


Изображение 2 - Схема взаимодействия рут-контракта и контракта свап-пары

Контракты рута свап-пары и свап-пары взаимодействуют следующим образом:

Рут-контракт выполняет деплой пары с выделением некоторого начального баланса TON-ов, который расходуется на деплой TIP-3 кошельков и некоторую первичную настройку: получение адресов кошельков из рут контрактов токена и задание адреса для колбэков. Рут-контракт также используется для обновления кода свап-пары.

Описание взаимодействия пара - токен

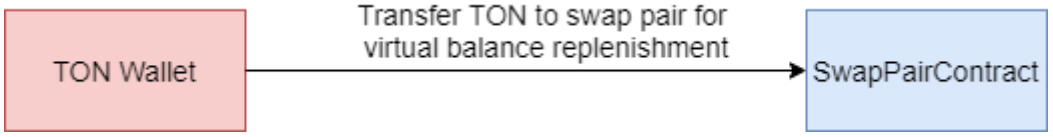


Изображение 3 - Описание взаимодействия TIP-3 кошелька и свап-пары. Красным отмечены взаимодействия, расходующие изначально выделяемый баланс свап-пары

Взаимодействие контракта свап-пары и токена заключается в переводе токенов пользователям и получении информации о переведённых токенах.

При переводе токенов на TIP-3 кошелёк, принадлежащий свап-паре, вызывается колбэк получения перевода. Перевод токенов пользователям инициируется в свап-паре с помощью вызова соответствующего метода, после чего происходит вызов TIP-3 кошелька из контракта свап-пары.

Описание взаимодействия TON кошелёк - пара



Изображение 4 - Взаимодействие TON кошелька и свап-пары

Взаимодействие контракта TON кошелька и свап-пары заключается в пополнении внутреннего баланса пользователя. Свап-пара поддерживает обработку внешних сообщений (так как они могут быть использованы при взаимодействии со свап-парой при помощи специальных сайтов или деботов), поэтому необходим внутренний баланс пользователя, который будет ограничивать взаимодействие пользователя и контракта пары, чтобы контракт не оказался в неактивном состоянии.

Механизм взимания платы за выполнение тех или иных действий описан в соответствующем разделе описания свап-пары.

При переводе необходимо приложить payload, содержащий публичный ключ пользователя в формате unit256, если публичный ключ получить не удалось, обратно отправляется только часть полученных TON-ов.

Сформировать необходимый payload можно следующим образом:

```
TvmBuilder builder;
builder.store(uint256(userPubkey));
TvmCell payload = builder.toCell();
```

Описание взаимодействия пользователь - пара

Сценарий взаимодействия пользователя и свап-пары выглядит следующим образом (при наличии уже существующей пары):

1. Пополнение своего виртуального баланса TON-ов в свап-паре для выполнения операций предоставления ликвидности, вывода ликвидности из пула и обмена токенов. Данный этап выполняется путём перевода TON-ов на контракт свап-пары с payload, содержащим публичный ключ пользователя;
2. Пополнение своего виртуального баланса токенов для свапа путём перевода токенов с TIP-3 кошелька на кошелёк свап-пары. В настоящий момент необходимо выполнять перевод с использованием функции **`transferWithNotify`**, чтобы контракт свап-пары был уведомлен о переводе;
3. Выполнение доступных операций с токенами: вклад/снятие ликвидности, обмен (swap);
4. Вывод токенов через запрос в свап-пару.

RootSwapPairContract

Краткое описание смарт-контракта

Данный смарт-контракт предназначен для создания свап-пар, хранения информации об уже существующих свап-парах, а также для обновления свап-пар при выходе новой версии.

Конструктор и static переменные

В контракте имеются следующие static переменные:

1. uint256 ownerPubkey - публичный ключ владельца рут-контракта свап-пар.

Конструктор на вход принимает следующие параметры:

1. TvmCell spCode - TvmCell, содержащий код контракта свап-пары;
2. uint32 spCodeVersion - версия кода контракта свап пары;
3. uint256 minMsgValue - минимальное количество TON-ов, прикреплённое к вызову функции для деплоя свап-пары;
4. uint256 contractSP - величина платы за сервисное обслуживание контракта, которая вычитается из стоимости входящего сообщения.

Функции

deploySwapPair - с помощью данной функции производится деплой свап-пары контракта. Конечный результат не зависит от порядка входных параметров - для вычисления адреса будущего контракта пары в качестве static переменной используется static поле uniqueId, вычисляемое как:

$$uniqueId = tokenRootContract.value \oplus tokenRootContract.value;$$

Что обеспечивает уникальность свап пары в рамках одного рут-контракта свапа.

На вход принимает следующие параметры:

1. address tokenRootContract1 - адрес рут-контракта токена;
2. address tokenRootContract2 - адрес рут-контракта токена.

Функция возвращает:

1. address cA - адрес задеплоенной свап-пары.

getPairInfo - данная функция используется для получения информации о существующих свап парах.

Аргументы функции:

1. address tokenRootContract1 - адрес рут-контракта токена 1;
2. address tokenRootContract2 - адрес рут-контракта токена 2.

Функция возвращает:

struct SwapPairInfo:

1. address rootContract - адрес рут контракта;
2. address tokenRoot1 - адрес рут-контракта токена 1;
3. address tokenRoot2 - адрес рут-контракта токена 2;
4. address tokenWallet1 - адрес кошелька токена 1 (равен нулевому адресу при получении информации из рут-контракта);
5. address tokenWallet2 - адрес кошелька токена 2 (равен нулевому адресу при получении информации из рут-контракта);
6. uint256 deployerPubkey - публичный ключ того, кто инициировал деплой свап-пары;
7. uint256 deployTimestamp - время деплоя свап-пары;
8. address swapPairAddress - адрес свап-пары;
9. uint256 uniqueId - уникальный номер пары;

10. uint32 swapPairCodeVersion - версия кода пары.

getServiceInformation - получение дополнительной информации о рут-контракте.

Аргументы функции: без аргументов
Функция возвращает:
struct ServiceInfo:
1. uint256 ownerPubkey - публичный ключ деплоера рут-контракта;
2. uint256 contractBalance - баланс в тонах;
3. uint256 creationTimestamp - время создания контракта;
4. uint32 codeVersion - версия кода свап-пары;
5. TvmCell swapPairCode - код свап-пары.

checkIfPairExists - функция для проверки существования свап-пары с заданными адресами рут-контрактов токенов.

Аргументы функции:
1. address tokenRootContract1 - адрес рут-контракта токена 1;
2. address tokenRootContract2 - адрес рут-контракта токена 2.

Функция возвращает:
1. bool value0 - существует ли свап-пара.

setSwapPairCode - функция для задания нового кода свап-пары на уровне рут-контракта. При попытке обновления пары будет использован код, хранящийся в паре. Функция может быть вызвана только тем, кто задеплоил изначальный рут-контракт пары.
Аргументы функции:
1. TvmCell code - новый код смарт-контракта пары;
2. uint32 codeVersion - версия кода смарт-контракта.

upgradeSwapPair - функция для обновления контракта свап-пары по её уникальному номеру с использованием имеющегося кода свап-пары. Может быть вызвана только тем, кто задеплоил свап-пару.

Аргументы функции:
1. uint256 uniqueID - уникальный номер пары.

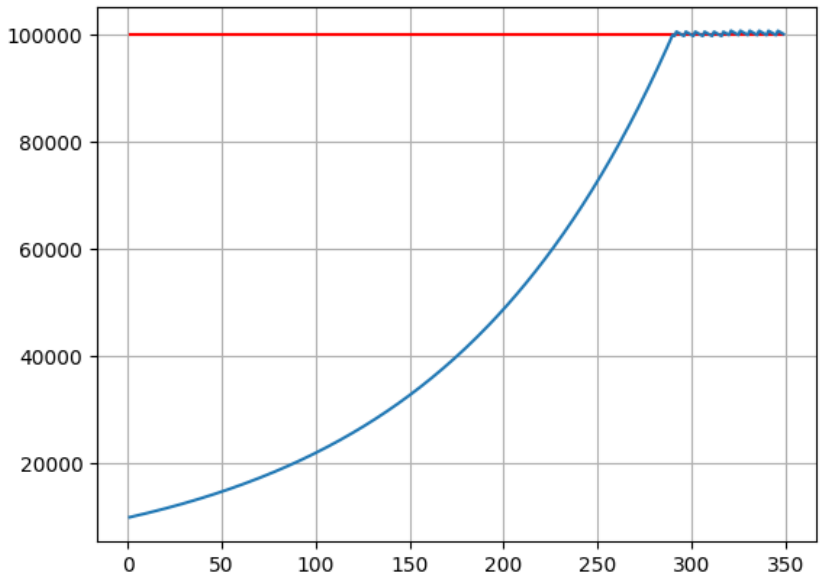
SwapPairContract

Краткое описание смарт-контракта

Данный смарт-контракт предназначен для реализации логики работы обмена на пулах ликвидности и предоставления пользователям доступа к данному функционалу.

Особенности функционирования смарт-контракта

Как уже ранее упоминалось, для функционирования контракта необходима плата за его выполнение. Для того, чтобы воспользоваться функционалом пулов ликвидности, пользователь должен предварительно внести TON-ы на свой виртуальный баланс свап-пары. Однако, встаёт вопрос, какой размер платы необходимо установить. Было решено реализовать механизм по балансировке платы за использование пулов ликвидности. Он работает следующим образом: сравнивается баланс контракта до и после выполнения “тяжёлых” функций, после чего выполняется изменение стоимости их выполнения. Стоимость возрастает быстрее чем убывает. Пример функционирования показан ниже на графике:



Изображение 5 - Функционирование механизма изменения стоимости выполнения

Как видно, стоимость довольно быстро сводится к средней стоимости выполнения функции, после чего начинает колебаться в небольших пределах относительно целевого значения стоимости выполнения.

Данный механизм позволяет не только динамически регулировать стоимость выполнения функций, но и также помогает реагировать на изменение баланса контракта, поддерживая его в рабочем состоянии.

Конструктор и static переменные

В контракте имеются следующие static переменные:

1. address token1 - адрес рут-контракта токена 1;
2. address token2 - адрес рут-контракта токена 2;
3. uint256 swapPairId - уникальный номер пары;

Конструктор на вход принимает следующие параметры:

1. address rootContract - адрес рут-контракта;
2. uint256 spd - публичный ключ инициатора деплоя контракта.

Функции

_reinitialize - в случае, если создание кошельков пары не было завершено, необходимо вызвать данную функцию для запуска повторной инициализации контракта. Вызвать функцию может только тот, кто деплоил свап-пару. К сообщению должны быть прикреплено как минимум 2 Ton-а.

Аргументы функции: без аргументов.

getCreationTimestamp - возвращает время создания свап-пары.

Функция возвращает:
1. uint256 value0 - время создания свап-пары.

getPairInfo - возвращает информацию о свап-паре.

Функция возвращает:
struct SwapPairInfo:
1. address rootContract - адрес рут-контракта;
2. address tokenRoot1 - адрес рут-контракта токена 1;
3. address tokenRoot2 - адрес рут-контракта токена 2;
4. address tokenWallet1 - адрес кошелька токена 1;
5. address tokenWallet2 - адрес кошелька токена 2;
6. uint256 deployerPubkey - публичный ключ создателя пары;
7. uint256 deployTimestamp - время создания пары.
8. address swapPairAddress - адрес свап-пары;
9. uint256 uniqueId - уникальный номер свап-пары;
10. uint32 swapPairCodeVersion - версия кода свап-пары.

getUserBalance - функция для получения баланса токенов пользователя, находящихся не в пуле ликвидности свап-пары. Необходим либо переданный параметр pubkey (для internal сообщений), либо подписанное сообщение и параметр pubkey = 0 (для external сообщений).

Аргументы функции:
1. uint256 pubkey - публичный ключ пользователя.

Функция возвращает:
struct UserBalanceInfo
1. address tokenRoot1 - адрес рут-контракта токена 1;
2. address tokenRoot2 - адрес рут-контракта токена 2;
3. uint128 tokenBalance1 - баланс пользователя в токене 1;
4. uint128 tokenBalance2 - баланс пользователя в токене 2.

getUserTONBalance - функция для получения баланса TON-ов пользователя в свап-паре. Необходим либо переданный параметр pubkey (для internal сообщений), либо подписанное сообщение и параметр pubkey = 0 (для external сообщений).

Аргументы функции:
1. uint256 pubkey - публичный ключ пользователя;

Функция возвращает:

1. uint256 balance - баланс пользователя в TOH-ах.

getUserLiquidityPoolBalance - функция для получения баланса токенов пользователя, находящихся в пуле ликвидности свап-пары. Необходим либо переданный параметр pubkey (для internal сообщений), либо подписанное сообщение и параметр pubkey = 0 (для external сообщений).

Аргументы функции:

1. uint256 pubkey - публичный ключ пользователя.

Функция возвращает:

struct UserBalanceInfo

1. address tokenRoot1 - адрес рут-контракта токена 1;
2. address tokenRoot2 - адрес рут-контракта токена 2;
3. uint256 userLiquidityTokenBalance - количество токенов, полученных в результате добавления ликвидности в пару (на текущий момент - некоторые условные единицы);
4. uint256 liquidityTokensMinted - общее количество выпущенных токенов (на текущий момент - условных единиц) пула ликвидности;
5. uint128 lpToken1 - количество токена 1 в пуле ликвидности;
6. uint128 lpToken2 - количество токена 2 в пуле ликвидности;

Процент владения пулом пользователем вычисляется по следующей формуле:

$userLPpercent = userLiquidityTokenBalance/liquidityTokensMinted;$

Количество токенов в пуле ликвидности, которыми владеет пользователь:

$userLPToken1 = userLPpercent * lpToken1;$

При вычислениях в смарт-контрактах рекомендуется воспользоваться функцией *math.muldiv* для избежания потери точности при вычислениях.

getExchangeRate - получение результата обмена заданного количества токенов.

Аргументы функции:

1. address swappableTokenRoot - выбранный токен для свапа;
2. uint128 swappableTokenAmount - заданное количество токенов для обмена.

Функция возвращает:

struct SwapInfo

1. uint128 swappableTokenAmount - количество токенов, использованное для свапа;
2. uint128 targetTokenAmount - количество токенов, которое может быть получено в результате свапа;
3. uint128 fee - комиссия за свап.

getProvidingLiquidityInfo - симуляция добавления токенов в пул ликвидности для определения результирующего количества добавленных токенов.

Аргументы функции:

1. uint128 maxFirstTokenAmount - максимальное количество токенов 1, внесённое в пул ликвидности;
2. uint128 masSecondTokenAmount - максимальное количество токенов 2, внесённое в пул ликвидности.

Функция возвращает:

1. uint128 providedFirstTokenAmount - количество токенов 1, которое было бы внесено в пару ликвидности;
2. uint128 providedSecondTokenAmount - количество токенов 2, которое было бы внесено в пару ликвидности.

getWithdrawLiquidityInfo - симуляция вывода токенов из пула ликвидности для определения результирующего количества выведенных токенов.
Аргументы функции:

1. uint128 minFirstTokenAmount - минимальное количество токенов 1 для вывода из пула ликвидности;
2. uint128 minSecondTokenAmount - минимальное количество токенов 2 для вывода из пула ликвидности;

Функция возвращает:

1. uint128 - withdrawedFirstTokenAmount - сколько токенов 1 было выведено из пула ликвидности;
2. uint128 - withdrawedSecondTokenAmount - сколько токенов 2 было выведено из пула ликвидности.

provideLiquidity - добавление токенов в пул ликвидности свап-пары.
Аргументы функции:

1. uint128 maxFirstTokenAmount - максимальное количество токенов 1, добавляемое в пул ликвидности;
2. uint128 maxSecondTokenAmount - максимальное количество токенов 2, добавляемое в пул ликвидности.

Функция возвращает:
struct SwapInfo

1. uint128 providedFirstTokenAmount - количество токенов 1, которое было внесено в пару ликвидности;
2. uint128 providedSecondTokenAmount - количество токенов 2, которое было внесено в пару ликвидности.

withdrawLiquidity - вывод токенов из пула ликвидности свап-пары.
Аргументы функции:

1. uint256 liquidityTokensAmount - количество виртуальных LP-токенов для вывода пропорционального количества токенов из пулов ликвидности.;

Функция возвращает:

1. uint128 withdrawedFirstTokenAmount - сколько токенов 1 было выведено из пула ликвидности;
2. uint128 withdrawedSecondTokenAmount - сколько токенов 2 было выведено из пула ликвидности.

swap - функция для обмена заданного количества выбранных токенов на другой токен.
Аргументы функции:

1. uint128 swappableTokenRoot - адрес рут-контракта выбранного токена для свапа;
2. uint128 swappableTokenAmount - количество токенов для обмена.

Функция возвращает:
struct SwapInfo

1. uint128 swappableTokenAmount - количество токенов, использованное для свапа;
2. uint128 targetTokenAmount - количество токенов, было получено в результате свапа;
3. uint128 fee - комиссия за свап, вычтенная из полученных токенов.

withdrawTokens - вывод заданного количества токенов из пары. Для вывода доступны токены, не находящиеся в данный момент в пуле ликвидности.
Аргументы функции:

1. address withdrawalTokenRoot - адрес рут-контракта выбранного токена;

2. address receiveTokenWallet - кошелёк для получения токенов;
3. uint128 amount - количество токенов для вывода.

getWalletAddressCallback - функция-колбэк для получения адреса кошелька токена. Может быть вызвана только рут-контрактами токенов пары.
Аргументы функции:

1. address walletAddress - адрес кошелька.

tokensReceivedCallback - функция-колбэк для получения информации о переводах токенов. Может быть вызвана только кошельками.

Аргументы функции:

1. address token_wallet - адрес кошелька токена;
2. address token_root - адрес рут-контракта токена;
3. uint128 amount - количество полученных токенов;
4. uint256 sender_public_key - публичный ключ отправителя токенов, который будет использован для добавления токенов на счёт пользователя;
5. address sender_address - адрес отправителя;
6. address sender_wallet - адрес кошелька-отправителя;
7. address original_gas_to - адрес, куда будет направлены излишки газа;
8. uint128 updated_balance - баланс кошелька после получения токенов;
9. TvmCell payload - прикреплённая дополнительная информация.

updateSwapPairCode - функция для обновления кода свап-пары. Может быть вызвана только рут-контрактом.

Аргументы функции:

1. TvmCell newCode - TvmCell, содержащий новый код контракта свап-пары;
2. uint32 newCodeVersion - новая версия кода контракта свап-пары.

Swap pair debot.

Общее описание

Для удобства взаимодействия пользователей с контрактом свап-пары, был разработан смарт-контракт дебота. Данный смарт-контракт значительно упрощает взаимодействие с контрактом свап-пары, так как не требует написания пользователем какого-либо кода. Единственным требованием при использовании дебота является предварительный перевод TON-ов и токенов на контракт свап-пары, так как данный дебот не поддерживает эти операции.

Функциональность Debot

С помощью данного дебота можно выполнять следующие операции:

1. Получение информации о доступных токенах для внесения в пул ликвидности или обмена;
2. Получение информации о токенах, находящихся в пуле ликвидности;
3. Получение информации о балансе пользователя в тонах;
4. Получение информации о стоимости выполнения функций, взаимодействующих с пулами ликвидности(добавление ликвидности, вывод ликвидности, обмен, вывод токенов из пары);
5. Добавление токенов в пул ликвидности;
6. Вывод токенов из пула ликвидности;
7. Получение текущего курса обмена;
8. Обмен токенов;
9. Вывод токенов из пары.