# Novaswap v2 Core

## Abstract

This technical whitepaper explains some of the design decisions behind the Novaswap core contracts. It covers the contracts' new features—including arbitrary pairs between ERC20s, a hardened price oracle that allows other contracts to estimate the time-weighted average price over a given interval, and a protocol fee that can be turned on in the future. It also re-architects the contracts to reduce their attack surface. This whitepaper describes the mechanics of Novaswap's "core" contracts including the pair contract that stores liquidity providers' funds—and the factory contract used to instantiate pair contracts.

## 1 Introduction

This paper describes the mechanics of that core contract, as well as the factory contract used to instantiate those contracts. Actually using Novaswap will require calling the pair contract through a "router" contract that computes the trade or deposit amount and transfers funds to the pair contract.

Contract code: https://github.com/novaswap-dev/Novaswap

## 2 New features

### 2.1 ERC-20 pairs

Novaswap allows liquidity providers to create pair contracts for any two ERC-20s.

A proliferation of pairs between arbitrary ERC-20s could make it somewhat more difficult to find the best path to trade a particular pair, but routing can be handled at a higher layer(either off-chain or through an on-chain router or aggregator).

### 2.2 Price oracle

The marginal price offered by Novaswap (not including fees) at time t can be computed by dividing the reserves of asset a by the reserves of asset b.

$$P_t = \frac{r_t^a}{r_t^b}$$

Since arbitrageurs will trade with Novaswap if this price is incorrect (by a sufficient amount to make up for the fee), the price offered by Novaswap tends to track the relative market price of the assets. This means it can be used as an approximate price oracle.

Novaswap improves this oracle functionality by measuring and recording the price before the first trade of each block (or equivalently, after the last trade of the previousblock). This price is more difficult to manipulate than prices during a block. If the attacker submits a transaction that attempts to manipulate the price at the end of a block,some other arbitrageur may be able to submit another transaction to trade back immediately afterward in the same block. A miner (or an attacker who uses enough gas to fill an entire block) could manipulate the price at the end of a block, but unless they mine the next block as well, they may not have a particular advantage in arbitraging the trade back.

Specifically, Novaswap accumulates this price, by keeping track of the cumulative sum of prices at the beginning of each block in which someone interacts with the contract. Each price is weighted by the amount of time that has passed since the last block in which it was updated, according to the block timestamp.2 This means that the accumulator value at any given time (after being updated) should be the sum of the spot price at each second in the history of the contract.

$$a_t = \sum_{i=1}^{t} p_i$$

To estimate the time-weighted average price from time t1 to t2, an external caller can checkpoint the accumulator's value at t1 and then again at t2, subtract the first value from the second, and divide by the number of seconds elapsed. (Note that the contract itself does not store historical values for this accumulator—the caller has to call the contract at the beginning of the period to read and store this value.)

$$p_{t1,t2} = \frac{\sum^{t2} i = t1 p_i}{t_2 - t_1} = \frac{\sum^{t2} i = t2 p_i - \sum^{t2} i = t1 p_i}{t_2 - t_1} = \frac{at_2 - a_{t_1}}{t_2 - t_1}$$

Users of the oracle can choose when to start and end this period. Choosing a longer period makes it more expensive for an attacker to manipulate the TWAP, although it results in a less up-to-date price.

One complication: should we measure the price of asset A in terms of asset B, or the price of asset B in terms of asset A? While the spot price of A in terms of B is always the reciprocal of the spot price of B in terms of A, the mean price of asset A in terms of asset B over a particular period of time is not equal to the reciprocal of the mean price of asset B in
terms of asset A.3 For example, if the USD/RNA price is 100 in block 1 and 300 in block 2, the average USD/RNA price will be 200 USD/RNA, but the average RNA/USD price will be 1/150 RNA/USD. Since the contract cannot know which of the two assets users would want to use as the unit of account, Novaswap tracks both prices.

Another complication is that it is possible for someone to send assets to the pair contract—and thus change its balances and marginal price—without interacting with it, andthus without triggering an oracle update. If the contract simply checked its own balances and updated the oracle based on the current price, an attacker could manipulate the oracle by sending an asset to the contract immediately before calling it for the first time in a
block. If the last trade was in a block whose timestamp was X seconds ago, the contract would incorrectly multiply the new price by X before accumulating it, even though nobodyhas had an opportunity to trade at that price. To prevent this, the core contract caches its reserves after each interaction, and updates the oracle using the price derived from the
cached reserves rather than the current reserves. In addition to protecting the oracle from manipulation, this change enables the contract re-architecture described below in section

## 2.2.1 Precision

Because Solidity does not have first-class support for non-integer numeric data types, the Novaswap uses a simple binary fixed point format to encode and manipulate prices. Specifically, prices at a given moment are stored as UQ112.112 numbers, meaning that 112 fractional bits of precision are specified on either side of the decimal point, with no sign. These numbers have a range of $[0, 2^{112} - 1]$ and a precision of $\frac{1}{2^{112}}$.
The UQ112.112 format was chosen for a pragmatic reason — because these numbers can be stored in a uint224, this leaves 32 bits of a 256 bit storage slot free. It also happens that the reserves, each stored in a uint112, also leave 32 bits free in a (packed) 256 bit storage slot. These free spaces are used for the accumulation process described above.Specifically, the reserves are stored alongside the timestamp of the most recent block with at least one trade, modded with 232 so that it fits into 32 bits. Additionally, although the price at any given moment (stored as a UQ112.112 number) is guaranteed to fit in 224 bits, the accumulation of this price over an interval is not. The extra 32 bits on the end of the storage slots for the accumulated price of A/B and B/A are used to store overflow bits resulting from repeated summations of prices. This design means that the price oracle only adds an additional three SSTORE operations (a current cost of about 15,000 gas) to the first trade in each block.
The primary downside is that 32 bits isn't quite enough to store timestamp values that will reasonably never overflow. In fact, the date when the Unix timestamp overflows a uint32 is 02/07/2106. To ensure that this system continues to function properly after this date, and every multiple of 232 – 1 seconds thereafter, oracles are simply required to check prices at least once per interval (approximately 136 years). This is because the core method of accumulation (and modding of timestamp), is actually overflow-safe, meaning that trades across overflow intervals can be appropriately accounted for given that oracles are using the proper (simple) overflow arithmetic to compute deltas.

## 2.3 Protocol fee

Novaswap includes a 0.05% protocol fee that can be turned on and off. If turned on, this fee would be sent to a feeTo address specified in the factory contract.Initially, feeTo is not set, and no fee is collected. A pre-specified address—feeToSetter—can call the setFeeTo function on the Novaswap factory contract, setting feeTo to a different value. feeToSetter can also call the setFeeToSetter to change the feeToSetter address itself.

If the feeTo address is set, the protocol will begin charging a 5-basis-point fee, which is taken as a 16 cut of the 30-basis-point fees earned by liquidity providers. That is, traders will continue to pay a 0.30% fee on all trades; 83.3% of that fee (0.25% of the amount traded) will go to liquidity providers, and 16.6% of that fee (0.05% of the amount traded) will go to the feeTo address.
Collecting this 0.05% fee at the time of the trade would impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned.
The total collected fees can be computed by measuring the growth in √
k since the last time fees were collected.6 This formula gives you the accumulated fees between t1 and t2 as a percentage of the liquidity in the pool at t2:

$$f_{1,2} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}}$$

If the fee was activated before t1, the feeTo address should capture 1/6 of fees that were accumulated between t1 and t2. Therefore, we want to mint new liquidity tokens to the feeTo address that represent $\phi * f_{1,2}$ of the pool, where $\phi = \frac{1}{6}$ .
That is, we want to choose sm to satisfy the following relationship, where s1 is the total quantity of outstanding shares at time t1:

$$\frac{s_m}{s_m + s_1} = \phi * f_{1,2}$$

After some manipulation, including substituting

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{\frac{1}{\phi} * \sqrt{k_2} + \sqrt{k_1}} * s_1$$

for f1,2 and solving for sm, we
can rewrite this as:

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{5 * \sqrt{k_2} + \sqrt{k_1}} * s_1$$

gives us the following formula:Suppose the initial depositor puts 100 USDT and 1 RNA into a pair, receiving 10 shares.Some time later (without any other depositor having participated in that pair), they attempt to withdraw it, at a time when the pair has 96 DAI and 1.5 ETH. Plugging those values into the above formula gives us the following:

$$s_m = 0.0286$$

## 2.4 Meta transactions for pool shares

Pool shares minted by Novaswap pairs natively support meta transactions. This means users can authorize a transfer of their pool shares with a signature, rather than an on-chain transaction from their address. Anyone can submit this signature on the user’s behalf by calling the permit function, paying gas fees and possibly performing other actions in the
same transaction.

# 3 Other changes

## 3.1 Solidity

Novaswap is implemented in the more widely-used Solidity.

## 3.2 Contract re-architecture

One design priority for Novaswap is to minimize the surface area and complexity of the core pair contract—the contract that stores liquidity providers’ assets. Any bugs in this contract could be disastrous, since millions of dollars of liquidity might be stolen or frozen.
When evaluating the security of this core contract, the most important question is whether it protects liquidity providers from having their assets stolen or locked. Any feature that is meant to support or protect traders—other than the basic functionality of allowing one asset in the pool to be swapped for another—can be handled in a "router" contract.
In fact, even part of the swap functionality can be pulled out into the router contract. As mentioned above, Novaswap stores the last recorded balance of each asset (in order to prevent a particular manipulative exploit of the oracle mechanism). The new architecture takes advantage of this to further simplify the contract.
In Novaswap, the seller sends the asset to the core contract before calling the swap function. Then, the contract measures how much of the asset it has received, by comparing the last recorded balance to its current balance. This means the core contract is agnosticto the way in which the trader transfers the asset. Instead of transferFrom, it could be a meta transaction, or any other future mechanism for authorizing the transfer of ERC-20s.

### 3.2.1 sync() and skim()

To protect against bespoke token implementations that can update the pair contract's balance, and to more gracefully handle tokens whose total supply can be greater than $2^{112}$, Novaswap has two bail-out functions: sync()and skim().
sync() functions as a recovery mechanism in the case that a token asynchronously deflates the balance of a pair. In this case, trades will receive sub-optimal rates, and if no liquidity provider is willing to rectify the situation, the pair is stuck. sync() exists to set the reserves of the contract to the current balances, providing a somewhat graceful recovery from this situation.
skim() functions as a recovery mechanism in case enough tokens are sent to an pair to overflow the two uint112 storage slots for reserves, which could otherwise cause trades to fail.
skim() allows a user to withdraw the difference between the current balance of the pair and $2^{112} - 1$ to the caller, if that difference is greater than 0.

## 3.3 Handling non-standard and unusual tokens

The ERC-20 standard requires that transfer() and transferFrom() return a boolean indicating the success or failure of the call. Novaswap interprets the missing return value of these improperly defined functions as false—that is, as an indication that the transfer was not successful—and reverts the transaction, causing the attempted transfer to fail. Novaswap handles non-standard implementations differently. Specifically, if a transfer() call has no return value, Novaswap interprets it as a success rather than as a failure. This change should not affect any ERC-20 tokens that conform to the standard (because in those tokens, transfer() always has a return value).
Novaswap also makes the assumption that calls to transfer() and transferFrom() cannot trigger a reentrant call to the Novaswap pair contract. This assumption is violated by certain ERC-20 tokens, including ones that support ERC-777's "hooks". To fully support such tokens, Novaswap includes a "lock" that directly prevents reentrancy to all public statechanging functions.

## 3.4 Initialization of liquidity token supply

When a new liquidity provider deposits tokens into an existing Novaswap pair, the number of liquidity tokens minted is computed based on the existing quantity of tokens:

$$s_{minted} = \frac{x_{deposited}}{x_{starting}} * s_{starting}$$

But what if they are the first depositor? In that case, $x_{starting}$ is 0, so this formula will not work. Novaswap sets the initial share supply to be equal to the amount of RNA deposited (in wei). This was a somewhat reasonable value, because if the initial liquidity was deposited at the correct price, then 1 liquidity pool share (which, like RNA, is an 18-decimal token) would be worth approximately 2 RNA. However, this meant that the value of a liquidity pool share was dependent on the ratio at which

liquidity was initially deposited, which was fairly arbitrary, especially since there was no guarantee that that ratio reflected the true price. Additionally, Novaswap supports arbitrary pairs, so many pairs will not include RNA at all.
Instead, Novaswap initially mints shares equal to the geometric mean of the amounts deposited:

$$s_{minted} = \sqrt{x_{deposited} * y_{deposited}}$$

This formula ensures that the value of a liquidity pool share at any time is essentially independent of the ratio at which liquidity was initially deposited. For example, suppose that the price of 1 ABC is currently 100 XYZ. If the initial deposit had been 2 ABC and 200 XYZ (a ratio of 1:100), the depositor would have received 20 shares. Those shares should now still be worth 2 ABC and 200 XYZ, plus accumulated fees.
If the initial deposit had been 2 ABC and 800 XYZ (a ratio of 1:400), the depositor would have received $\sqrt{2 \cdot 800} = 40$ pool shares.
The above formula ensures that a liquidity pool share will never be worth less than the geometric mean of the reserves in that pool. However, it is possible for the value ofa liquidity pool share to grow over time, either by accumulating trading fees or through "donations" to the liquidity pool. In theory, this could result in a situation where the value of the minimum quantity of liquidity pool shares (1e-18 pool shares) is worth so much that it becomes infeasible for small liquidity providers to provide any liquidity.
To mitigate this, Novaswap burns the first 1e-15 (0.000000000000001) pool shares that are minted (1000 times the minimum quantity of pool shares), sending them to the zero address instead of to the minter. This should be a negligible cost for almost any token pair. But it dramatically increases the cost of the above attack. In order to raise the value of a liquidity pool share to $100, the attacker would need to donate 100,000 to the pool, which would be permanently locked up as liquidity.

## 3.5 Wrapping RNA

The interface for transacting with GeneChain's native asset, RNA, is different from the standard interface for interacting with ERC-20 tokens: using a canonical wrapped RNA token `WRNA`. Native RNA needs to be wrapped into `WRNA` before it can be traded on Novaswap.

## 3.6 Deterministic pair addresses

Novaswap uses Genechain’s new CREATE2 opcode to generate a pair contract with a deterministic address. This means that it is possible to calculate a pair’s address (if it exists) off-chain, without having to look at the chain state.

## 3.7 Maximum token balance

In order to efficiently implement the oracle mechanism, Novaswap only support reserve balances of up to $2^{112} - 1$. This number is high enough to support 18-decimal-place tokens with a totalSupply over 1 quadrillion.
If either reserve balance does go above $2^{112} - 1$, any call to the swap function will begin to fail (due to a check in the _update() function). To recover from this situation, any user can call the skim() function to remove excess assets from the liquidity pool.

# 4 NOVA token

## 4.1 Allocation

`NOVA` is the Novaswap protocol token. `NOVA` releases 100,000,000

1. 20% to investors and will be released befor Novaswap is going live, 10% by pre-sale, 10% by airdrop.
2. 20% to team members and future employees with 4-year vesting, and 1/16 released every quarter.
3. 60% as a liquidity mining reward pool

- Each trade pair to be launched on swap has a reward plan, with a specified period and a specified total number of NOVA rewards, which are rewarded to accounts who provide liquidity and distributed according to the proportion.
- The plan to launch swap trade pairs will be announced in advance.
- A perpetual inflation rate of 2% or 1,200,000 per year for the pool.

## 4.2 Liquidity Mining

An initial liquidity mining program will go live one week after the Novaswap app is launched.

# 5 Governance

Initial governance parameters are as follows:

- 1% of NOVA total supply (delegated) to submit a governance proposal
- 4% of NOVA supply required to vote ’yes’ to reach quorum
- 7 day voting period
- 2 day timelock delay on execution