Niftyswap Specification

** Certain sections of this document were taken directly from the Uniswap documentation.*

Table of Content

Overview

Niftyswap is a fork of Uniswap, a protocol for automated token exchange on Ethereum. While Uniswap is for trading ERC-20 tokens, Niftyswap is a protocol for ERC-1155 tokens. Both are designed to favor ease of use

and provide guaranteed access to liquidity on-chain.

Most exchanges maintain an order book and facilitate matches between buyers and sellers. Niftyswap smart contracts hold liquidity reserves of various tokens, and trades are executed directly against these reserves. Prices are set automatically using the constant product $x*y = K$ market maker mechanism, which keeps overall reserves in relative equilibrium. Reserves are pooled between a network of liquidity providers who supply the system with tokens in exchange for a proportional share of transaction fees.

An important feature of Niftyswap is the utilization of a factory/registry contract that deploys a separate exchange contract for each ERC-1155 token contract. These exchange contracts each hold independent reserves of a single fungible ERC-1155 currency and their associated ERC-1155 token id. This allows trades between the Currency and the ERC-1155 tokens based on the relative supplies.

This document outlines the core mechanics and technical details for Niftyswap.

Contracts

NiftyswapExchange.sol

This contract is responsible for permitting the exchange between a single currency and all tokens in a given ERC-1155 token contract. For each token id i, the NiftyswapExchance contract holds a reserve of currency and a reserve of token id i, which are used to calculate the price of that token id i denominated in the currency.

NiftyswapFactory.sol

This contract is used to deploy a new NiftyswapExchange.sol contract for ERC-1155 contracts without one yet. It will keep a mapping of each ERC-1155 token contract address with their corresponding NiftyswapExchange.sol contract address.

Contract Interactions

All methods should be free of arithmetic overflows and underflows.

Methods for exchanging tokens and managing reserves liquidity are all called internally via the ERC-1155 `onERC1155BatchReceived()` method. The four methods that can be called via `onERC1155BatchReceived()` should be safe against re-entrancy attacks.

```
/**
 * @notice Handle which method is being called on transfer
 * @dev `_data` must be encoded as follow: abi.encode(bytes4, MethodObj)
 *   where bytes4 argument is the MethodObj signature passed as defined
 *   in the `Signatures for onReceive control logic` section above
 * @param _operator The address which called safeBachTransferFrom()
 * @param _from     The address which previously owned the Token
 * @param _ids      An array containing Token ids being transferred
 * @param _amounts  An array containing token amounts being transferred
 * @param _data     Method signature and corresponding encoded arguments
 * @return bytes4(keccak256(
 *  "onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)"
 * ))
 */
function onERC1155BatchReceived(
```

```
  address _operator,
  address _from,
  uint256[] memory _ids,
  uint256[] memory _amounts,
  bytes memory _data)
  public returns(bytes4);
```

The first 4 bytes of the `_data` argument indicate which of the four main NiftyswapExchange.sol methods to call. How to build and encode the `_data` payload for the respective methods is explained in the Data Encoding section.

Exchanging Tokens

In `NiftyswapExchange.sol`, there are two methods for exchanging tokens:

```
/**
 * @notice Convert currency tokens to Tokens _id and transfers Tokens to recipient.
 * @dev User specifies MAXIMUM inputs (_maxCurrency) and EXACT outputs.
 * @dev Assumes that all trades will be successful, or revert the whole tx
 * @dev Exceeding currency tokens sent will be refunded to recipient
 * @dev Sorting IDs is mandatory for efficient way of preventing duplicated IDs (which
would lead to exploit)
 * @param _tokenIds             Array of Tokens ID that are bought
 * @param _tokensBoughtAmounts  Amount of Tokens id bought for each token id in
_tokenIds
 * @param _maxCurrency          Total maximum amount of currency tokens to spend
 * @param _deadline             Block number after which this transaction will be
reverted
 * @param _recipient            The address that receives output Tokens and refund
 */
function _currencyToToken(
  uint256[] memory _tokenIds,
  uint256[] memory _tokensBoughtAmounts,
  uint256 _maxCurrency,
  uint256 _deadline,
  address _recipient)
  internal nonReentrant();

/**
 * @notice Convert Tokens _id to currency tokens and transfers Tokens to recipient.
 * @dev User specifies EXACT Tokens _id sold and MINIMUM currency tokens received.
 * @dev Assumes that all trades will be valid, or the whole tx will fail
 * @dev Sorting _tokenIds is mandatory for efficient way of preventing duplicated IDs
(which  would lead to errors)
 * @param _tokenIds          Array of Token IDs that are sold
 * @param _tokensSoldAmounts Array of Amount of Tokens sold for each id in _tokenIds.
 * @param _minCurrency       Minimum amount of currency tokens to receive
 * @param _deadline          Block number after which this transaction will be
reverted
 * @param _recipient         The address that receives output currency tokens.
 */
function _tokenToCurrency(
```

```
  uint256[] memory _tokenIds,
  uint256[] memory _tokensSoldAmounts,
  uint256 _minCurrency,
  uint256 _deadline,
  address _recipient)
  internal nonReentrant();
```

Managing Reserves Liquidity

In `NiftyswapExchange.sol`, there are two methods for managing token reserves supplies:

```
/**
 * @notice Deposit less than max currency tokens && exact Tokens (token ID) at current
ratio to mint liquidity pool tokens.
 * @dev min_liquidity does nothing when total liquidity pool token supply is 0.
 * @dev Assumes that sender approved this contract on the currency
 * @dev Sorting _tokenIds is mandatory for efficient way of preventing duplicated IDs
(which  would lead to errors)
 * @param _provider      Address that provides liquidity to the reserve
 * @param _tokenIds      Array of Token IDs where liquidity is added
 * @param _tokenAmounts  Array of amount of Tokens deposited for each ID in _tokenIds
 * @param _maxCurrency   Array of maximum number of tokens deposited for ids in
_tokenIds.
 *                       Deposits max amount if total liquidity pool token supply is
0.
 * @param _deadline      Block number after which this transaction will be reverted
 */
function _addLiquidity(
  address _provider,
  uint256[] memory _tokenIds,
  uint256[] memory _tokenAmounts,
  uint256[] memory _maxCurrency,
  uint256 _deadline)
  internal nonReentrant();

/**
 * @dev Burn liquidity pool tokens to withdraw currency  && Tokens at current ratio.
 * @dev Sorting _tokenIds is mandatory for efficient way of preventing duplicated IDs
 * @param _provider         Address that removes liquidity to the reserve
 * @param _tokenIds         Array of Token IDs where liquidity is removed
 * @param _poolTokenAmounts Array of Amount of liquidity tokens burned for ids in
_tokenIds.
 * @param _minCurrency      Minimum currency withdrawn for each Token id in _tokenIds.
 * @param _minTokens        Minimum Tokens id withdrawn for each Token id in
_tokenIds.
 * @param _deadline         Block number after which this transaction will be reverted
 */
function _removeLiquidity(
  address _provider,
  uint256[] memory _tokenIds,
  uint256[] memory _poolTokenAmounts,
  uint256[] memory _minCurrency,
```

```
    uint256[] memory _minTokens,
    uint256 _deadline)
    internal nonReentrant()
```

Price Calculations

In Niftyswap, like Uniswap, the price of an asset is a function of a currency reserve and the corresponding token reserve. Indeed, all methods in Niftyswap enforce that the following equality remains true:

$CurrencyReserve_i * TokenReserve_i = K$

where $CurrencyReserve_i$ is the currency reserve size for the corresponding token id i, $TokenReserve_i$ is the reserve size of the ERC-1155 token id i and K is an arbitrary constant.

Ignoring the Liquidity Fee, purchasing some tokens i with the currency (or vice versa) should increase the $CurrencyReserve_i$ and decrease the $TokenReserve_i$ (or vice versa) such that

$CurrencyReserve_i * TokenReserve_i == K$.

Determining the cost of *purchasing* $\Delta{}TokenReserve_i $ tokens i therefore depends on the quantity purchased, such that

$\Delta{}CurrencyReserve_i = \frac{K}{TokenReserve_i - \Delta{}TokenReserve_i} - CurrencyReserve_i$

with substitution, the purchase cost can also be written as

$\Delta{}CurrencyReserve_i = \frac{CurrencyReserve_i * \Delta{}TokenReserve_i}{TokenReserve_i - \Delta{}TokenReserve_i} $

where $\Delta{}CurrencyReserve_i$ is the amount of currency tokens that must be sent cover the cost of the $\Delta{}TokenReserve_i $ purchased. The latter form of this equation is the one used in the `getBuyPrice()` function. Inversely, determining the revenue from *selling* $\Delta{}TokenReserve_i $ tokens i can be done with

$\Delta{}CurrencyReserve_i = CurrencyReserve_i - \frac{K}{TokenReserve_i + \Delta{}TokenReserve_i}$

with substitution, the purchase cost can also be written as

$\Delta{}CurrencyReserve_i = \frac{CurrencyReserve_i * \Delta{}TokenReserve_i}{TokenReserve_i + \Delta{}TokenReserve_i}$

where $\Delta{}CurrencyReserve_i$ is the amount of currency that a user would receive. The latter form of this equation is the one used in the `getSellPrice()` function.

Note that the implementation of these equations is subjected to arithmetic rounding errors. To see how these are mitigated, see the Rounding Errors section.

#Liquidity Fee

A liquidity provider fee of **0.5%** paid in the currency is added to every trade, increasing the corresponding $CurrencyReserve_i$. Compared to the 0.3% fee chosen by Uniswap, the 0.5% fee was chosen to ensure that token reserves are deep, which ultimately provides a better experience for users (less slippage, better price discovery, and lower risk of transactions failing). This value could change for Niftyswap V2.

While the $CurrencyReserve_i$ / $TokenReserve_i$ ratio is constantly shifting, fees makes sure that the total combined reserve size increases with every trade. This functions as a payout to liquidity providers that is collected when they burn their liquidity pool tokens to withdraw their portion of total reserves.

This fee is asymmetric, unlike with Uniswap, which will bias the ratio in one direction. However, one the bias becomes large enough, an arbitrage opportunity will emerge and someone will correct that bias. This leads to some inefficiencies, but this is necessary as some ERC-1155 tokens are non-fungible (0 decimals) and the fees can only be paid with the currency. Note that highly illiquid 0 decimal tokens could have issues when it comes to withdrawing liquidity, due to rounding errors.

Assets

Within Niftyswap, there are two main types of assets: the **currency** and the **tokens**. While the currency is also expected to be an ERC-1155 token, this document always refers to the ERC-1155 token that acts as currency as the "currency", while using the tokens that are traded against the currency are referred to as "tokens" or "token i" to indicate an arbitrary token with an id i.

Currency

The currency is an ERC-1155 token that is fungible (>0 decimals) and is used to price each token i in a given ERC-1155 token contract. For instance, this currency could be wrapped Ether or wrapped DAI (see erc20-meta-wrapper). Since the currency is an ERC-1155, the NiftyswapExchange.sol contract also needs to be aware of what the currency `id` is in the currency contract. The currency can be the same contract as the Tokens contract.

Both the address and the token id of the currency can be retrieved by calling getCurrencyInfo(). Note that if the currency and the tokens are the same ERC-1155 contract, the `currencyID <=> currencyID` pool will be prohibited for security reasons.

Tokens

The tokens contract is an ERC-1155 compliant contract where each of its token ids are priced with respect to the currency. These tokens *can* have 0 decimals, meaning that some token ids are not divisible. The liquidity provider fee accounts for this possibly as detailed in the Liquidity Fee section. **Note that 0 decimal tokens can face issues if highly illiquid when it comes to removing liquidity.**

The address of the ERC-1155 token contract can be retrieved by calling getTokenAddress().

Trades

All trades are done by specifying exactly how many tokens i a user wants to buy or sell, without exactly knowing how much currency they will send or receive. This design choice was necessary considering ERC-1155 tokens can be non-fungible, unlike the currency which is assumed to be fungible (non-zero decimals). All trades will update the corresponding currency and token reserves correctly and will be subjected to a liquidity provider fee.

It is possible to buy/sell multiple tokens at once, but if any one fails, the entire trade will fail as well. This could change for Niftyswap V2.

Currency to Token i

To trade currency => token i, a user would call

```
_currencyToToken(_tokenIds, _tokensBoughtAmounts, _maxCurrency, _deadline,
_recipient);
```

as defined in the Exchanging Tokens section and specify *exactly* how many tokens i they expect to receive from the trade. This is done by specifying the token ids to purchase in the `_tokenIds` array and the amount for each token id in the `_tokensBoughtAmounts` array.

Since users can't know exactly how much currency will be required when the transaction is created, they must provide a `_maxCurrency` value which contain the maximum amount of currency they are willing to spend for the entire trade. It would have been possible for Niftyswap to support a maximum amount per token i, however this would increase the gas cost significantly. If this feature is proven to be desired it could be incorporated in Niftyswap V2.

Additionally, to protect users against miners or third party relayers withholding their Niftyswap trade transactions, a `_deadline` parameter must be provided by the user. This `_deadline` is a block number after which a given transaction will revert.

Finally, users can specify who should receive the tokens with the `_recipient` argument. This is particularly useful for third parties and proxy contracts that will interact with Niftyswap.

The `_maxCurrency` argument is specified as the amount of currency sent to the NiftyswapExchange.sol contract via the `onERC1155BatchReceived()` method :

```
// Tokens received need to be currency contract
require(msg.sender == address(currency), "NiftyswapExchange#onERC1155BatchReceived:
INVALID_CURRENCY_TRANSFERRED");
require(_ids.length == 1, "NiftyswapExchange#onERC1155BatchReceived:
INVALID_CURRENCY_IDS_AMOUNT");
require(_ids[0] == currencyID, "NiftyswapExchange#onERC1155BatchReceived:
INVALID_CURRENCY_ID");

// Decode BuyTokensObj from _data to call _currencyToToken()
BuyTokensObj memory obj;
(, obj) = abi.decode(_data, (bytes4, BuyTokensObj));
address recipient = obj.recipient == address(0x0) ? _from : obj.recipient;

// Buy tokens
_currencyToToken(obj.tokensBoughtIDs, obj.tokensBoughtAmounts, _amounts[0],
obj.deadline, recipient);
```

where any difference between the actual cost of the trade and the amount sent will be refunded to the specified recipient.

To call this method, users must transfer sufficient currency to the NiftyswapExchange.sol, as follow:

```
// Call _currencyToToken() on NiftyswapExchange.sol contract
IERC1155(CurrencyContract).safeTranferFrom(_from, niftyswap_address, curency_id,
_maxCurrency, _data);
```

where `_data` is defined in the Data Encoding: _currencyToToken() section.

Token i to Currency

To trade token i => currency, a user would call

```
_tokenToCurrency(_tokenIds, _tokensSoldAmounts, _minCurrency, _deadline, _recipient);
```

as defined Exchanging Tokens and specify *exactly* how many tokens i they sell. This is done by specifying the token ids to sell in the `_tokenIds` array and the amount for each token id in the `_tokensSoldAmounts` array.

Since users can't know exactly how much currency they would receive when the transaction is created, they must provide a `_minCurrency` value which contain the minimum amount of currency they are willing to accept for the entire trade. It would have been possible for Niftyswap to support a minimum amount per token i, however this would increase the gas cost significantly. If proven to be desired, this could be incorporated in Niftyswap V2.

Additionally, to protect users against miners or third party relayers withholding their Niftyswap trade transactions, a `_deadline` parameter must be provided by the user. This `_deadline` is a block number after which a given transaction will revert.

Finally, users can specify who should receive the currency with the `_recipient` argument upon the completion of the trade. This is particularly useful for third parties and proxy contracts that will interact with Niftyswap.

The `_tokenIds` and `_tokensSoldAmounts` arguments are specified as the token ids and token amounts sent to the NiftyswapExchange.sol contract via the `onERC1155BatchReceived()` method :

```
// Tokens received need to be correct ERC-1155 Token contract
require(msg.sender == address(token), "NiftyswapExchange#onERC1155BatchReceived:
INVALID_TOKENS_TRANSFERRED");

// Decode SellTokensObj from _data to call _tokenToCurrency()
SellTokensObj memory obj;
(functionSignature, obj) = abi.decode(_data, (bytes4, SellTokensObj));
address recipient = obj.recipient == address(0x0) ? _from : obj.recipient;

// Sell tokens
_tokenToCurrency(_ids, _amounts, obj.minCurrency, obj.deadline, recipient);
```

To call this method, users must transfer the tokens to sell to the NiftyswapExchange.sol contract, as follow:

```
// Call _tokenToCurrency() on NiftyswapExchange.sol contract
IERC1155(TokenContract).safeBatchTranferFrom(_from, niftyswap_address, _ids, _amounts,
_data);
```

where `_data` is defined in the Data Encoding: _tokenToCurrency() section.

Liquidity Reserves Management

Anyone can provide liquidity for a given token i, so long as they also provide liquidity for the corresponding currency reserve. When adding liquidity to a reserve, liquidity providers should not influence the price, hence the contract ensures that calling `_addLiquidity()` or `_removeLiquidity()` does not change the $ CurrencyReserve_i / TokenReserve_i $ ratio.

Adding Liquidity

To add liquidity for a given token i, a user would call

```
_addLiquidity(_provider, _tokenIds, _tokenAmounts, _maxCurrency, _deadline);
```

as defined in Managing Reserves Liquidity section. Similarly to trading, when adding liquidity, users specify the exact amount of token i without knowing the exact amount of currency to send. This is done by specifying the token ids to sell in the `_tokenIds` array and the amount for each token id in the `_tokenAmounts` array.

Since users can't know exactly how much currency will be required when the transaction is created, they must provide a `_maxCurrency` array which contains the maximum amount of currency they are willing to add as liquidity for each token i.

Additionally, to protect users against miners or third party relayers withholding their Niftyswap trade transactions, a `_deadline` parameter must be provided by the user. This `_deadline` is a block number after which a given transaction will revert.

The `_provider` argument is the address of who sent the tokens and the `_tokenIds` and `_tokenAmounts` arguments are specified as the token ids and token amounts sent to the NiftyswapExchange.sol contract via the `onERC1155BatchReceived()` method:

```
// Tokens received need to be correct ERC-1155 Token contract
require(msg.sender == address(token), "NiftyswapExchange#onERC1155BatchReceived:
INVALID_TOKEN_TRANSFERRED");

// Decode AddLiquidityObj from _data to call _addLiquidity()
AddLiquidityObj memory obj;
(functionSignature, obj) = abi.decode(_data, (bytes4, AddLiquidityObj));

// Add Liquidity
_addLiquidity(_from, _ids, _amounts, obj.maxCurrency, obj.deadline);
```

To call this method, users must transfer the tokens to add to the NiftyswapExchange.sol liquidity pools, as follow:

```
// Call _addLiquidity() on NiftyswapExchange.sol contract
IERC1155(TokenContract).safeBatchTranferFrom(_provider, niftyswap_address, _ids,
_amounts, _data);
```

where `_data` is defined in the Data Encoding: _addLiquidity() section.

Removing Liquidity

To remove liquidity for a given token i, a user would call

```
_removeLiquidity(_provider, _tokenIds, _poolTokenAmounts, _minCurrency, _minTokens,
_deadline);
```

as defined in Managing Reserves Liquidity section. Users must specify *exactly* how many liquidity pool tokens they want to burn. This is done by specifying the token ids to sell in the `_tokenIds` array and the amount for each token id in the `_poolTokenAmounts` array.

Since users can't know exactly how much currency and tokens they will receive back when the transaction is created, they must provide a `_minCurrency` and `_minTokens` arrays, which contain the minimum amount of currency and tokens i they are willing to receive when removing liquidity.

Additionally, to protect users against miners or third party relayers withholding their Niftyswap trade transactions, a `_deadline` parameter must be provided by the user. This `_deadline` is a block number after which a given transaction will revert.

The `_provider` argument is the address of who sent the liquidity pool tokens, the `_tokenIds` and `_poolTokenAmounts` arguments are specified as the token ids and liquidity pool token amounts sent to the NiftyswapExchange.sol contract via the `onERC1155BatchReceived()` method:

```
// Tokens received need to be NIFTY-1155 tokens (liquidity pool tokens)
require(msg.sender == address(this), "NiftyswapExchange#onERC1155BatchReceived:
INVALID_NIFTY_TOKENS_TRANSFERRED");

// Decode RemoveLiquidityObj from _data to call _removeLiquidity()
RemoveLiquidityObj memory obj;
(functionSignature, obj) = abi.decode(_data, (bytes4, RemoveLiquidityObj));

// Remove Liquidity
_removeLiquidity(_from, _ids, _amounts, obj.minCurrency, obj.minTokens, obj.deadline);
```

To call this method, users must transfer the liquidity pool tokens to burn to the NiftyswapExchange.sol contract, as follow:

```
// Call _removeLiquidity() on NiftyswapExchange.sol contract
IERC1155(NiftyswapExchange).safeBatchTranferFrom(_provider, niftyswap_address, _ids,
_amounts, _data);
```

where `_data` is defined in the Data Encoding: _removeLiquidity() section.

Data Encoding

In order to call the correct NiftySwap method, users must encode a data payload containing the function signature to call and the method's receptive argument objects. All method calls must be encoded as follow:

```
// bytes4 method_signature
// Obj method_struct
_data = abi.encode(method_signature, method_struct);
```

where the `method_signature` and `method_struct` are specific to each method. The `_data` argument is then passed in the `safeBatchTransferFrom(..., _data)` function call.

_currencyToToken()

The `bytes4` signature to call this method is `0x24c186e7`

```
// bytes4(keccak256(
//   "_currencyToToken(uint256[],uint256[],uint256,uint256,address)"
// ));
bytes4 internal constant BUYTOKENS_SIG = 0xb2d81047;
```

The `method_struct` for this method is structured as follow:

Elements	Type	Description

recipient	address	Who receives the purchased tokens
tokensBoughtIDs	uint256[]	Token IDs to buy
tokensBoughtAmounts	uint256[]	Amount of token to buy for each ID
deadline	uint256	Block # after which the tx isn't valid anymore

or

```
struct BuyTokensObj {
    address recipient;             // Who receives the tokens
    uint256[] tokensBoughtIDs;     // Token IDs to buy
    uint256[] tokensBoughtAmounts; // Amount of token to buy for each ID
    uint256 deadline;              // Block # after which the tx isn't valid anymore
}
```

_tokenToCurrency()

The `bytes4` signature to call this method is `0xdb08ec97`

```
// bytes4(keccak256(
//   "_tokenToCurrency(uint256[],uint256[],uint256,uint256,address)"
// ));
bytes4 internal constant SELLTOKENS_SIG = 0x7db38b4a;
```

The `method_struct` for this method is structured as follow:

Elements	Type	Description
recipient	address	Who receives the currency for the sale
minCurrency	uint256	Minimum number of currency expected for all tokens sold
deadline	uint256	Block # after which the tx isn't valid anymore

or

```
struct SellTokensObj {
    address recipient;    // Who receives the currency for the sale
    uint256 minCurrency;  // Minimum number of currency expected for all tokens sold
    uint256 deadline;     // Block # after which the tx isn't valid anymore
}
```

_addLiquidity()

The `bytes4` signature to call this method is `0x82da2b73`

```
//  bytes4(keccak256(
//   "_addLiquidity(address,uint256[],uint256[],uint256[],uint256)"
// ));
bytes4 internal constant ADDLIQUIDITY_SIG = 0x82da2b73;
```

The `method_struct` for this method is structured as follow:

Elements	Type	Description
maxCurrency	uint256[]	Maximum number of currency to deposit with tokens
deadline	uint256	Block # after which the tx isn't valid anymore

or

```
struct AddLiquidityObj {
    uint256[] maxCurrency; // Maximum number of currency to deposit with tokens
    uint256 deadline;      // Block # after which the tx isn't valid anymore
}
```

_removeLiquidity()

The `bytes4` signature to call this method is `0x5c0bf259`

```
// bytes4(keccak256(
//    "_removeLiquidity(address,uint256[],uint256[],uint256[],uint256[],uint256)"
// ));
bytes4 internal constant REMOVELIQUIDITY_SIG = 0x5c0bf259;
```

The `method_struct` for this method is structured as follow:

Elements	Type	Description
minCurrency	uint256[]	Minimum number of currency to withdraw
minTokens	uint256[]	Minimum number of tokens to withdraw
deadline	uint256	Block # after which the tx isn't valid anymore

or

```
struct RemoveLiquidityObj {
    uint256[] minCurrency; // Minimum number of currency to withdraw
    uint256[] minTokens;   // Minimum number of tokens to withdraw
    uint256 deadline;      // Block # after which the tx isn't valid anymore
}
```

Relevant Methods

These methods are useful for clients and third parties to query the current state of a NiftyswapExchange.sol contract.

getCurrencyReserves()

```
function getCurrencyReserves(
    uint256[] calldata _ids
) external view returns (uint256[] memory)
```

This method returns the amount of currency in reserve for each Token i in `_ids`.

getPrice_currencyToToken()

```
function getPrice_currencyToToken(
    uint256[] calldata _ids,
    uint256[] calldata _tokensBoughts
) external view returns (uint256[] memory)
```

This method will return the current cost for the token _ids provided and their respective amount.

getPrice_tokenToCurrency()

```
function getPrice_tokenToCurrency(
    uint256[] calldata _ids,
    uint256[] calldata _tokensSold
) external view returns (uint256[] memory)
```

This method will return the current amount of currency to be received for the token _ids and their respective amount in `tokensSold`.

getTokenAddress()

```
function tokenAddress() external view returns (address);
```

Will return the address of the corresponding ERC-1155 token contract.

getCurrencyInfo()

```
function getCurrencyInfo() external view returns (address, uint256);
```

Will return the address of the currency contract that is used as currency and its corresponding id.

Miscellaneous

Rounding Errors

Some rounding errors are possible due to the nature of finite precision arithmetic the Ethereum Virtual Machine (EVM) inherits from. To account for this, some corrections needed to be implemented to make sure these rounding errors can't be exploited.

Three main functions in NiftyswapExchange.sol are subjected to rounding errors: `_addLiquidity()`, `_currencyToToken()` and `_tokenToCurrency()`.

For `_addLiquidity()`, the rounding error can occur at

```
uint256 currencyAmount = tokenAmount.mul(currencyReserve) / tokenReserve.sub(amount);
```

where `currencyAmount` is the amount of currency that needs to be sent to NiftySwap for the given `tokenAmount` of token i added to the liquidity. Rounding errors could lead to a smaller value of `currencyAmount` than expected, favoring the new liquidity provider, hence we add `1` to the amount that is required to be sent if a rounding error occurred.

Inversely, if a rounding error occurred when calculating the `currencyAmount`, the amount of liquidity tokens to be minted will favor the new liquidity provider instead of existing liquidity providers, which is

undesirable. To compensate, we calculate the amount of liquidity token to mint to new liquidity provider as follow ;

```
liquiditiesToMint[i] = (currencyAmount.sub(rounded ? 1 : 0)).mul(totalLiquidity) /
currencyReserve
```

For `_currencyToToken()` , the rounding error can occur at

```
// Calculate buy price of card
uint256 numerator = _currencyReserve.mul(_tokenBoughtAmount);
uint256 denominator = (_tokenReserve.sub(_tokenBoughtAmount));
uint256 cost = numerator / denominator;
```

where `cost` is the amount of currency that needs to be sent to NiftySwap for the given `_tokenBoughtAmount` of token i being purchased. Rounding errors could lead to a smaller value of `cost` than expected, favoring the buyer, hence we add `1` to the amount that is required to be sent if a rounding error occurred.

For `_tokenToCurrency()` , the rounding error can occur at

```
// Calculate sell price of card
uint256 numerator = _tokenSoldAmount.mul(_currencyReserve);
uint256 denominator = _tokenReserve.add(_tokenSoldAmount);
uin256 revenue = numerator / denominator;
```

where `revenue` is the amount of currency that will to be sent to buyer for the given `_tokenSoldAmount` of token i being sold. Rounding errors could lead to a smaller value of `revenue` than expected, disfavoring the buyer, hence no correction is necessary if rounding error occurs.

Notably, **rounding errors and the applied correction only have a significant impact when the currency use has a low number of decimals.**