Tinyman AMM V2 Protocol Specification

Overview

Tinyman AMM V2 is a Constant Product Market Maker. It is mathematically similar to Tinyman V1 which in turn was modeled on Uniswap V2. The contract and protocol design for Tinyman AMM V2 is significantly different from V1 as it uses more recent features of the Algorand Virtual Machine.

An AMM based decentralized exchange is made up of a number of ‘Pools’, with one pool per asset pair. Users can swap assets through these pools by sending some of one asset and receiving the equivalent value of the other asset.

The funds in the pools are owned by ‘Liquidity Providers’. Each liquidity provider contributes and retains ownership of a share of the pool’s funds. The liquidity provider receives ‘Pool Tokens’ to represent their share of the pool. They can reclaim their assets at any time by returning the pool tokens.

Each Pool is an Algorand Account. The accounts hold the assets and state related to the pool. A single stateful Algorand Application contains all the logic that controls the asset transfers and state management of the pools.

Each Pool has a number of local state variables which are updated during every operation. The asset reserves (liquidity) of the pool are tracked in the local state and not determined dynamically from the account balances. This means that ‘donations’ to the pool or Algo making up the minimum balance do not contribute to the asset reserves.

If assets are transferred to the pool account with transactions that are not part of a documented group the assets will be lost. These assets are called “extra” and are only transferable to the fee collector. They will not affect the liquidity of the pool or the shares of the liquidity providers.

Pool Creation

Pools are created with two separate operations; Bootstrap & Add Initial Liquidity.

A pool is created as a Smart Signature Contract Account. The Smart Signature is derived from a standard template but it is unique per pool because it contains the ids of the pair of assets.

Bootstrap

The account is turned into a pool by issuing an Application Call transaction to the AMM app. This transaction opts the account into the app and also rekeys the contract account to the App account.
As a contract account, this transaction must be signed by the associated Smart Signature. After this transaction is confirmed the Smart Signature no longer has authority over the account as it has been rekeyed. Instead the AMM App has full authority over the pool account.

During the Bootstrap operation the app creates the Pool Token asset and transfers the total supply to the pool account. These are done with Inner Transactions. All pool token assets are created by the app account so that there is a single well known creator address of Tinyman AMM V2 pool tokens. The Reserve address of the pool token asset is set to the pool address.

The Bootstrap operation requires some Algo to cover the minimum balances for the pool account (Formula I), asset creation (100000 micro Algo) and transaction fees. This Algo must be supplied to the pool prior to the Bootstrap Application Call. It will usually be in the same group but this is not required.

Transaction Details for bootstrap

Adding Liquidity

Funds are added to the pool with Asset Transfer transactions together with an Application Call. Pool tokens are sent to the user by the pool using an inner transaction. There are two methods for adding liquidity.

Adding Initial Liquidity

If the pool has no liquidity then the first user to add liquidity is free to set the ratio between the two assets. If the pool has any liquidity this transaction fails and the user should use the *Adding Subsequent Liquidity* method. The formula to determine the number of pool tokens for initial liquidity is Formula II.A.

Transaction Details for add_initial_liquidity

Adding Subsequent Liquidity

The formula to determine the number of pool tokens for subsequent liquidity is Formula II.B.

The protocol supports adding liquidity in a flexible way so that the user does not have to provide amounts of assets at exactly the current pool ratio. Instead, it supports a continuum between liquidity at exactly the current ratio and all liquidity in a single asset. This is achieved by internal implicit swaps where necessary. A swap fee is charged on the amount that must be swapped. The value of this fee is reduced from the pool tokens output.

If the user does provide liquidity at exactly the current ratio then there will be no swap fee and pool tokens will be output for the entirety of the liquidity provided.

A special case of providing liquidity in a flexible way is providing liquidity in only one single asset. In this case the user does not need to transfer a zero amount but can instead omit the transfer transaction entirely.

When providing liquidity at a ratio different from the current pool ratio the user must be aware of the loss of value due to both price impact and fees of the implicit swap. The user can pre-calculate the expected value of pool tokens accounting for these and provide the min_output parameter to ensure the amount of pool tokens output matches their expectations. This can protect the user from sudden price changes between the time of signing and executing the transaction.

Adding liquidity with two assets

When adding liquidity with two assets two transfer transactions are required in addition to the application call transaction. The 2nd application arg must be the byte string “flexible” in this case.

Adding liquidity with a single asset

When adding liquidity with one asset one transfer transaction is required in addition to the application call transaction. The 2nd application arg must be the byte string “single” in this case.

Transaction Details for add_liquidity

Removing Liquidity

Liquidity providers can remove funds from the pool with an Asset Transfer of pool tokens and an Application Call.

Amounts of each asset are sent to the user by the pool using two inner transactions. The amount of each asset received is proportional to the pool tokens returned and dependent on the current ratio of the pool. See Formula III.A.

Removing liquidity as a single asset

The protocol supports removing liquidity as a single asset from pools by combining removing liquidity with an internal swap calculation. The entirety of one side of the liquidity will be swapped into the other asset. A swap fee is charged on this amount and deducted from the output.
See Formula III.B.

When using this convenience method the user must be aware of the loss of value due to both price impact and fees of the implicit swap. The user can pre-calculate the expected output accounting for these and provide the min_output parameter to ensure the amount of

output matches their expectations. This can protect the user from sudden price changes between the time of signing and executing the transaction.

Swapping

Swaps are made with an Asset Transfer transaction together with an Application Call.
An amount of the other asset is sent to the user by the pool using an inner transaction.
The type of swap may be either fixed-input or fixed-output. If the type is fixed-output an extra inner transaction is sent to return any change from the input amount to the user (see below).
A fixed-output swap has a higher transaction fee because it requires one extra inner transaction.

A swap fee is charged on all swaps. The fee is set per pool as basis points. This fee is deducted from the input amount before applying the swap formula to calculate the output.

The formulae for fixed-input and fixed output swaps are described in Formula IV.A & IV.B respectively.

Slippage Tolerance

Swaps have slippage tolerance to allow for changes to the exchange rate between the time the swap was prepared and the swap was executed. For a fixed-input swap a minimum output amount is included in the Application Call arguments. If the calculated output amount is less than this amount the swap will fail.

For a fixed-output swap the slippage tolerance is included in the input amount and the exact output amount is specified in the arguments. If the calculated input amount required to achieve the target output amount is less than the provided input amount the swap fails. If it is greater then the change is returned to the user by an inner transaction.

Transaction Details for swap

Fees

Tinyman charges fees on swaps, loans and internal swap operations. A portion of this fee is allocated to the protocol (Tinyman) and the remainder is allocated to the liquidity providers by adding to the pool reserves.

The fee rate and the protocol fee fraction portion are configurable parameters on each pool. The total fee rate can be set in the range from 0.01% to 1% (10 to 100 basis points), the protocol fee fraction can be set as 1/n in the range from 1/3 to 1/10. The default fee rate is %0.3 and protocol fee fraction is 1/6 (0.05% of total input). These defaults are the same as the Tinyman V1.

Pools have a single fee setting and the same rate is applied to all swap, loan and internal swap operations.

The protocol fees accumulate in both assets. They can be claimed in full at any time by any user that issues an appropriate Application Call. The accumulated amounts of both assets are transferred by inner transactions to the fee collector. The fee collector is specified by a global state variable of the App (fee_collector). The fee_collector can be set by the global fee_manager address.

Fee parameters can be set by the global fee_setter address. The fee_setter address can be set by the global fee_manager address. The fee_manager address can be set by the current fee_manager address.

Swap Fees

Swap fees are collected from the input assets.

Examples:

A. Fixed Input
   If the input amount is 5000A, fee is *5000*fee rate* AssetA. If the fee rate is 0.3%, total fee is 15 AssetA, pooler fee is 13 AssetA and protocol fee is 2 AssetA.

   If the input amount is 20000A, fee is 20000A*fee rate* AssetA. If the fee rate is 0.3%, total fee is 60 AssetA and protocol fee is 10 AssetA.

B. Fixed Output
   In this method, the extra amount added because of the slippage is returned back to the users with an inner transaction. The input amount is calculated by subtracting the extra amount from the transferred amount and the fee is applied to this amount.

   If the transferred amount is 1005A, and extra amount is 5, the fee is applied to 1000 AssetA.

Flash Loan Fees

The fee rate is the same with the swap fee rate of the pool. It is applied to the loan amounts. The repayment must be equal or greater than the total of loan amount and fee.

Examples:

A. Loan single asset
   If the loan is 3000 AssetA, fee is *3000*fee rate* AssetA. It is 9 AssetA if the fee rate is 0.3%, so minimum repayment amount is 3009 AssetA.

B. Loan multiple assets
   If the loan is 3000 AssetA and 5000 AssetB, fee is *3000*fee rate* AssetA and *5000*fee rate* AssetB. The minimum repayment amounts are 3009 AssetA and 5015 AssetB, if the fee rate is 0.3%.

Flash Swap Fees

The fee calculation is the same with swap. The same swap fee rate is applied to input amounts.

Examples:

A. Pay in single asset
   If the payment is 3000 AssetA, the fee is *3000*fee rate* AssetA.

B. Pay in multiple assets
   If the payment is 1000 AssetA and 2000 AssetB, the fees are *1000*fee rate* AssetA and *2000*fee rate* AssetB.

Flash Loan

Flash loan allows users to borrow single or multiple assets with no collateral with the condition of paying it back within the same transaction group. It allows advanced users to create a custom transaction group to generate a profit without an initial capital.

Flash loan requires two application calls. The first one is for requesting a loan from the pool and the second one is for verifying the repayment.

The repayments must be made in the same asset. The repayment amount should cover the loan amount and fee. The swap fee rate applies to loan amounts.

With the first app call users specify the amount of loans in terms of asset 1 and asset 2 and group index difference between first and second application calls. Loan amounts cannot exceed the pool reserves. Requested assets are transferred to the user using inner transactions.

Verification calls check the repayment amounts. The transaction fails if they don't cover the loan amount and fee. Repayments should be placed just before the verification app call and the contract checks the transfer amounts. Any extra payments are considered donations to pools.

Flash loan transactions can appear anywhere within a transaction group and a single group may contain multiple flash loans.

Users can use the same pool freely between the two application calls and do a swap. Unlike Flash Swap (below) usage of the pool is not restricted.

Flash Swap

Flash swap has similarities to swap and flash loan. It is designed for advanced users. They can create a custom transaction group, borrow assets from the pool with an application call and make a profit (arbitrage, collateral swap etc.), pay it back within the same transaction group and make another application call to verify the payment. If the payment amount is insufficient all transactions in the atomic group fail. It allows users to generate profit without risk and collateral.

Unlike the flash loan, users are free to pay back in both assets as long as the final pool invariant increases and covers the fee. This can allow for more efficient transactions in some scenarios compared to the Flash Loan.

In the regular swap, the user transfers input assets to the pool first and then receives the output assets. Flash swap allows users to change this order. They can receive the output assets first, and use these assets to make a profit, and transfer input assets to the pool. In this scenario, the applied fee is exactly the same as the swap and the fee is applied to input assets and collected in terms of input assets. If users receive 500 AssetA and send 1000 AssetB, fee amount will be 3 AssetB with default fee rate 0.3% (1000*0.003=3).

Additionally, similar to flash loan, flash swap allows users to borrow a single asset or both assets of the pool. Unlike the flash loan, users are free to pay back in both assets as long as the final pool invariant increases and covers the fee. Fees are applied to all input assets. If the repayment is made as 1000 AssetA and 2000 AssetB, the fee is collected in both assets as 3 AssetA and 6 AssetB. If the repayment is made in a single asset such as 3000 AssetB, the fee is 9 AssetB.

With the first app call, users specify the amount of loans in terms of asset 1 and asset 2 and group index difference between first and second (verification) application calls. Loan amounts cannot exceed the pool reserves. Requested assets are transferred to the user using inner transactions.

The verification calls check the initial and final invariant. Repayments must be made between two app calls. The app checks the final balance to determine the paid amounts and doesn't check a particular transfer so users can make multiple inner and outer transactions. Any extra payments are considered as donations to the pools.

The pool is locked between the two application calls and no operation (swap, add & remove liquidity etc.) is allowed. These operations can be used before or after the flash swap in the same transaction group however.

Additional Notes

Donations

Any assets that are in the pool accounts but not part of the reserves, protocol fees or minimum Algo balance are claimable to the fee_collector address.

Algorand Participation Rewards that accumulate in accounts holding Algo are considered donations to the pool and do not contribute to the pool reserves in AMM V2 (unlike V1). As of 2022/05/14 the Participation Rewards program is finished and no further rewards are planned to be distributed by this mechanism. In the event that this mechanism is re-enabled in the future the pool accounts would accumulate rewards that would be claimable by the fee_collector address. It would be up to the protocols managers (development team or DAO) to decide how those rewards should be used.

Algo Pools

The AMM V2 protocol supports ASA-Algo pools just like V1. All references to ‘Assets’ in this document equally apply to Algo. The transaction details differ slightly whenever Algo is involved as Payment transactions are used instead of Asset Transfer.

Transaction Fees

All transaction fees must be paid by the sender of any of the outer transactions of an operation. The inner transaction fees cannot be paid by the pool account and app account.

Users

This document refers to ‘Users’ as the senders of assets and transactions to the pool and receivers of assets from the pool. The protocol does not place any restriction on the type of account that initiates a pool operation and therefore supports operations made by application accounts in the same way as regular accounts. Additionally there is no requirement that the sender of all transactions in an operation group is the same account.
The receiver of assets (Pool Tokens, swapped assets, removed assets, change, etc) is always the sender of the Application Call.

Transaction Groups & Composability

Each operation described above requires multiple transactions that must be part of the same atomic group. The exact indices of the transactions within the group or the size of the group is not specified by the protocol however so it is possible to create groups containing additional transactions before or after a swap for example. However relative indexing is used so the relative location of transactions within the group is specified and must be adhered to.

Operation transaction groups may also be created as inner transactions from within another Application. The only exception to this is the Bootstrap operation for creating pools, which requires a Smart Signature that cannot be used from an inner transaction.

Pool Discovery

Clients can discover the existence of pools in either of two ways;

1) Every pool account is opted into the Tinyman App. Any account opted into this app is a valid pool. A list of pool accounts can therefore be retrieved from an Algorand Indexer or other block follower. The local state of these accounts gives the details of the pool.
2) The pool account address can be computed from the LogicSig template.

Permissioned Methods / Manager Functionality

The protocol includes a number of permissioned methods. These are methods that can only be called by specific addresses. In this section we describe what these methods can and cannot do.

Roles

The fee_manager, fee_setter and fee_collector are distinct roles in the protocol. Each of these are mutable.

Fee Collector

The fee_collector is expected to be a single signature account used by an off-chain system to actively manage the accumulated protocol fees and extras. There is a not insignificant chance that this account could become compromised so in order to protect future earnings of the protocol this account must be replaceable by a higher authority - the fee_manager.

Fee Setter

The fee_setter is expected to initially be a multisig account but over time it may transition to a system account for an automated process or a contract account governed by voting. To facilitate these changes of use and to protect against compromised keys the fee_setter is configurable only by the fee_manager.

Fee Manager

The fee_manager is expected to be a multisig account initially, and potentially a contract account governed by voting in the future. It is assumed that this account is never compromised. The protocol requires that the role can be transferred between accounts.

The fee_manager, fee_setter and fee_collector roles are all initialized to the creator address of the application. This is assumed to be a multisig.

These roles are global - they cannot be defined separately for different pools.

There are no other roles or management capabilities defined in the protocol. There is no capability to pause/lock/close/empty any pool and there is no capability to upgrade the application.

Methods

Set Fee Collector

This method is callable by the `fee_manager`. It updates the global state value of `fee_collector` to a new address.

Set Fee Setter

This method is callable by the `fee_manager`. It updates the global state value of `fee_setter` to a new address.

Set Fee

This method is callable by the `fee_setter`. It updates the pool local state values of `total_fee_share` and `protocol_fee_ratio` to new values. The protocol enforces limits on the allowable values for each of these. This allows the swap fees of each pool to

vary over time without moving or fragmenting liquidity. It is intended that these changes will be made based on sound market analysis for the benefit of the poolers and protocol.

Set Fee Manager

This method is callable by the `fee_manager`. It updates the global state value of `fee_manager` to a new address.

Related Non Permissioned Methods

Claim Fees

This method is callable by any address. It transfers all accumulated protocol fees to the `fee_collector` account. This operation does not affect the liquidity of the pool.

Claim Extra

This method is callable by any address. It allows transfering the extra asset and algo amounts from application or pool accounts to the `fee_collector` account. This operation does not affect the liquidity of the pool. “extra amounts” are defined as amounts held in the pool account that are not part of the asset reserves, protocol fees or minimum balance requirement.

Formulae

I. Pool Minimum Balance

Ref: [Minimum balance requirement for a smart contract]

Local state uint count: 12
Local state byte slice count: 2

A. ASA-ASA Pool

```
100000 (Accounts on Algorand require a minimum balance of 100,000
micro Algo)
+ 100000 ASA 1 (Asset Optin)
+ 100000 ASA 2 (Asset Optin)
+ 100000 Pool Token (Asset Optin)
+ 542000 App Optin (100000 + (25000+3500)*12 + (25000+25000)*2)
= 942000 micro Algo
```

B. ASA-ALGO Pool

```
100000 (Accounts on Algorand require a minimum balance of 100,000
micro Algo)
+ 100000 ASA 1 (Asset Optin)
~~+ 100,000 ASA 2 (Asset Optin)~~
+ 100000 Pool Token (Asset Optin)
+ 542000 App Optin (100000 + (25000+3500)*12 + (25000+25000)*2)
= 842000 micro Algo
```

II. Adding Liquidity

A. Adding Initial Liquidity

$$Issued\ Pool\ Tokens\ =\ floor(\sqrt{Asset1\ Amount\ *\ Asset2\ Amount})$$
$$Locked\ Pool\ Tokens\ =\ 1000$$
$$Pool\ Token\ Output\ =\ Issued\ Pool\ Tokens\ -\ Locked\ Pool\ Tokens$$

B. Adding Subsequent Liquidity

The same formula applies to single and flexible modes.

$$Old\ K\ =\ Asset1\ Reserves\ *\ Asset2\ Reserves$$
$$New\ K\ =\ (Asset1\ Reserves\ +\ Asset1\ Amount)\ *\ (Asset2\ Reserves\ +\ Asset2\ Amount)$$
$$R\ =\ \sqrt{Old\ K}\ /\ Issued\ Pool\ Tokens$$
$$New\ Issued\ Pool\ Tokens\ =\ \sqrt{New\ K}\ /\ R$$
$$New\ Issued\ Pool\ Tokens\ =\ \sqrt{New\ K}\ /\ (\sqrt{Old\ K}\ /\ Issued\ Pool\ Tokens)$$
$$New\ Issued\ Pool\ Tokens\ =\ \sqrt{(New\ K\ *\ Issued\ Pool\ Tokens\ *\ Issued\ Pool\ Tokens)\ /\ Old\ K}$$

$$Pool\ Tokens\ Out\ =\ New\ Issued\ Pool\ Tokens\ -\ Old\ Issued\ Pool\ Tokens$$
$$New\ Asset1\ Reserves\ =\ Old\ Asset1\ Reserves\ +\ Asset1\ Amount$$
$$New\ Asset2\ Reserves\ =\ Old\ Asset1\ Reserves\ +\ Asset2\ Amount$$

$$Calculated\ Asset1\ Amount\ =\ floor((Pool\ Tokens\ Out\ *\ New\ Asset1\ Reserves)\ /\ New\ Issued\ Pool\ Tokens)$$
$$Calculated\ Asset2\ Amount\ =\ floor((Pool\ Tokens\ Out\ *\ New\ \ Asset2\ Reserves)\ /\ New\ Issued\ Pool\ Tokens)$$
$$Asset1\ Swap\ Amount\ =\ Asset1\ Amount\ -\ Calculated\ Asset1\ Amount$$
$$Asset2\ Swap\ Amount\ =\ Asset2\ Amount\ -\ Calculated\ Asset2\ Amount$$
$$Swap\ Amount\ =\ Max(Asset1\ Swap\ Amount,\ Asset2\ Swap\ Amount)$$

$$Total\ Fee\ Amount\ =\ (Swap\ Amount\ *\ Total\ Fee\ Share)\ /\ (10000\ -\ Total\ Fee\ Share)$$
$$Protocol\ Fee\ Amount\ =\ Total\ Fee\ Amount\ /\ Protocol\ Fee\ Ratio$$
$$Poolers\ Fee\ Amount\ =\ Total\ Fee\ Amount\ -\ Protocol\ Fee\ Amount$$

$Asset1\ Swap\ Amount\ >\ Asset2\ Swap\ Amount$:

$$Fee\ As\ Pool\ Tokens\ =\ (Total\ Fee\ Amount\ *\ New\ Issued\ Pool\ Tokens)\ /\ (New\ Asset1\ Reserves\ *\ 2)$$
$$New\ Asset1\ Reserves\ =\ New\ Asset1\ Reserves\ -\ Protocol\ Fee\ Amount$$
$$Pool\ Tokens\ Out\ =\ Pool\ Tokens\ Out\ -\ Fee\ As\ Pool\ Tokens$$

$Asset2\ Swap\ Amount\ >\ Asset1\ Swap\ Amount$:

$$Fee\ As\ Pool\ Tokens\ =\ (Total\ Fee\ Amount\ *\ New\ Issued\ Pool\ Tokens)\ /\ (New\ Asset2\ Reserves\ *\ 2)$$
$$New\ Asset2\ Reserves\ =\ New\ Asset2\ Reserves\ -\ Protocol\ Fee\ Amount$$
$$Pool\ Tokens\ Out\ =\ Pool\ Tokens\ Out\ -\ Fee\ As\ Pool\ Tokens$$

III. Removing Liquidity

A. Multiple Asset Out

a. Remove all circulating pool tokens

$$Issued\ Pool\ Tokens\ ==\ Locked\ Pool\ Tokens\ +\ Removed\ Pool\ Tokens$$
$$Asset1\ Output\ =\ Asset1\ Reserves$$
$$Asset2\ Output\ =\ Asset2\ Reserves$$

b. Otherwise

$$Issued\ Pool\ Tokens\ >\ Locked\ Pool\ Tokens\ +\ Removed\ Pool\ Tokens$$
$$Asset1\ Output\ =\ floor(Removed\ Pool\ Token\ Amount\ *\ Asset1\ Reserves\ /\ Issued\ Pool\ Tokens)$$
$$Asset2\ Output\ =\ floor(Removed\ Pool\ Token\ Amount\ *\ Asset2\ Reserves\ /\ Issued\ Pool\ Tokens)$$

B. Single Asset Out

Calculate the $Asset1\ Output$ and $Asset2\ Output$ as multiple assets out and convert an asset to desired output by doing an internal swap (fixed-input swap).

a. Asset 1 Out

$$Input\ Amount\ =\ Asset2\ Output$$
$$Swap\ Output\ =\ FixedInputSwap(Input\ Amount)$$
$$Asset1\ Output\ =\ Asset1\ Output\ +\ Swap\ Output$$
$$Asset2\ Output\ =\ 0$$

b. Asset 2 Out

$$Input\ Amount\ =\ Asset1\ Output$$
$$Swap\ Output\ =\ FixedInputSwap(Input\ Amount)$$
$$Asset2\ Output\ =\ Asset2\ Output\ +\ Swap\ Output$$
$$Asset1\ Output\ =\ 0$$

IV. Swapping

A. Fixed Input

$$Total\ Fee\ =\ floor((Input\ Amount\ *\ Total\ Fee\ Share)\ /\ 10000)$$
$$Protocol\ Fee\ =\ floor(Total\ Fee\ /\ Protocol\ Fee\ Ratio)$$
$$Poolers\ Fee\ =\ Total\ Fee\ -\ Protocol\ Fee$$
$$Swap\ Amount\ =\ Input\ Amount\ -\ Total\ Fee$$
$$K\ =\ Input\ Supply\ *\ Output\ Supply$$
$$Swap\ Output\ =\ Output\ Supply\ -\ (floor(K\ /\ (Input\ Supply\ +\ Swap\ Amount))\ +\ 1)$$

B. Fixed Output

$$Swap\ Amount\ =\ (floor(K\ /\ (Output\ Supply\ -\ Output\ Amount))\ +\ 1)\ -\ Input\ Supply$$
$$Input\ Amount\ =\ floor((Swap\ Amount\ *\ 10000)\ /\ (10000\ -\ Total\ Fee\ Share))$$
$$Total\ Fee\ =\ Input\ Amount\ -\ Swap\ Amount$$
$$Protocol\ Fee\ =\ floor(Total\ Fee\ /\ Protocol\ Fee\ Ratio)$$
$$Poolers\ Fee\ =\ Total\ Fee\ -\ Protocol\ Fee$$
$$Change\ =\ Input\ Transfer\ Amount\ -\ Input\ Amount$$

V. Flash Loan

Applies to both assets.

$$Total\ Fee\ =\ floor((Loan\ Amount\ *\ Total\ Fee\ Share)\ /\ 10000)$$
$$Protocol\ Fee\ =\ floor(Total\ Fee\ /\ Protocol\ Fee\ Ratio)$$
$$Poolers\ Fee\ =\ Total\ Fee\ -\ Protocol\ Fee$$
$$Expected\ Repayment\ Amount\ =\ Loan\ Amount\ +\ Total\ Fee$$
$$Donation\ Amount\ =\ Repayment\ Transfer\ Amount\ -\ Expected\ Repayment\ Amount$$

VI. Flash Swap

Applies to both assets.

$$Input\ Amount\ =\ Transfer\ Amount\ (Final\ Balance\ -\ Initial\ Balance)$$

Note: The rest is the same with fixed input swap.

$$Total\ Fee\ =\ floor((Input\ Amount\ *\ Total\ Fee\ Share)\ /\ 10000)$$
$$Protocol\ Fee\ =\ floor(Total\ Fee\ /\ Protocol\ Fee\ Ratio)$$
$$Poolers\ Fee\ =\ Total\ Fee\ -\ Protocol\ Fee$$

Protocol Limits

1. Assets must have a minimum total of 1,000,000. Rounding errors of 1 microunit are expected in the calculations. To avoid unexpected losses of significant value, this limit is enforced at pool bootstrap.

2. Swap operations with no fee are not allowed. If the swap amount is too low, the calculated fee can be 0 because of the integer arithmetic. The swap will fail in this case.

   Example calculations for fixed input:
   a. 0.3% Fee -> ceil(10000 / 30) = 334 minimum input amount
   b. 0.1% Fee -> ceil(10000 / 10) = 100 minimum input amount
   c. 1% Fee -> ceil(10000 / 100) = 10 minimum input amount
3. Maximum Input Amounts

   Swap:

   Fixed input:
   $floor(18446744073709551615 \ / \ Total\ Fee\ Share)$

   Fixed output:
   $floor(18446744073709551615 \ / \ 10000)$

   Flash Loan, maximum loan amount:
   $floor(18446744073709551615 \ / \ Total\ Fee\ Share)$

   Flash Swap, maximum loan amount:
   $floor(18446744073709551615 \ / \ Total\ Fee\ Share)$

Protocol Methods

This section describes the transaction groups for each method of the protocol.

Where more than one transaction is required they must appear in an atomic group. The exact indices within the group are not important unless specified. However the relative positions of transactions for each method are important. The transaction order may not change and no other transactions may appear between the transactions of each method.

Where *user_address* is specified the same address must be used for all instances within a method. *any_address* can be any value.

Bootstrap

Transactions

Pay:

Sender: any_address
Receiver: pool_address
Amount:
Pool minimum balance (Formula I)
+ Transaction fee of the app call
+ 100000 (min transaction fee for asset creation)

AppCall:

Sender: pool_address (Logic Sig)
Index: tinyman_amm_v2_app_id
OnComplete: OptIn
App Args: [“bootstrap”]
Foreign Assets: [asset_1_id, asset_2_id]
RekeyTo: tinyman_amm_v2_app_address
Fee: (7 * min_fee) (Note: 6 * min_fee if asset 2 is Algo)

Side Effects

Local State Changes

1. pool_token_asset_id
2. asset_1_id
3. asset_2_id
4. total_fee_share
5. protocol_fee_ratio
6. asset_1_reserves
7. asset_2_reserves
8. issued_pool_tokens
9. asset_1_protocol_fees
10. asset_2_protocol_fees
11. lock
12. asset_1_cumulative_price

13. asset_2_cumulative_price
14. cumulative_price_update_timestamp

Inner Transactions

1. **Pay:**
   Sender: pool_address
   Receiver: tinyman_amm_v2_app_address
   Amount: 100_000
   Fee: 0

2. **AssetConfig:**
   Sender: tinyman_amm_v2_app_address
   UnitName: “TMPOOL2”
   Name: “TinymanPool2.0 {asset_1_unit_name}-{asset_2_unit_name}”
   MetadataHash: “{asset_1_id}{asset_2_id}{trailing zeros}”
   uint64 (8 bytes) + uint64 (8 bytes) + zeros (16 bytes)
   Total: 18446744073709551615
   Reserve: pool_address
   Decimals: 6
   URL: “https://tinyman.org”
   Fee: 0

3. **AssetTransfer (Opt-In):**
   Sender: pool_address
   Receiver: pool_address
   Index: asset_1_id
   Amount: 0

4. **AssetTransfer (Opt-In) (If Asset 2 is not Algo):**
   Sender: pool_address
   Receiver: pool_address
   Index: asset_2_id
   Amount: 0

5. **AssetTransfer (Opt-In):**
   Sender: pool_address
   Receiver: pool_address
   Index: created_asset_id
   Amount: 0

6. **AssetTransfer:**
   Sender: tinyman_amm_v2_app_address
   Receiver: pool_address
   Index: created_asset_id
   Amount: 18446744073709551615

Add Initial Liquidity

Transactions

1. **AssetTransfer:**
   Sender: user_address
   Receiver: pool_address
   Index: asset_1_id
   Amount: asset_1_amount

2. **AssetTransfer/Pay:**
   Sender: user_address
   Receiver: pool_address
   Index: asset_2_id
   Amount: asset_2_amount

3. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“add_initial_liquidity”]
   Foreign Assets: [pool_token_asset_id]
   Accounts: [pool_address]
   Fee: (2 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. issued_pool_tokens

Inner Transactions

1. **AssetTransfer:**
   Sender: pool_address
   Receiver: user_address
   Index: pool_token_asset_id
   Amount: *(See formula II.A for pool token calculation)*

Add Subsequent Liquidity

A. Flexible

Transactions

1. **AssetTransfer:**

Sender: user_address
Receiver: pool_address
Index: asset_1_id
Amount: asset_1_amount

2. **AssetTransfer/Pay:**
   Sender: user_address
   Receiver: pool_address
   Index: asset_2_id
   Amount: asset_2_amount

3. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“add_liquidity”, “flexible”, min_output]
   Foreign Assets: [pool_token_asset_id]
   Accounts: [pool_address]
   Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. issued_pool_tokens
4. asset_1_protocol_fees
5. asset_2_protocol_fees
6. asset_1_cumulative_price
7. asset_2_cumulative_price
8. cumulative_price_update_timestamp

Logs

1. input_asset_id
2. output_asset_id
3. swap_amount
4. poolers_fee_amount
5. protocol_fee_amount
6. total_fee_amount

Inner Transactions

1. **App Call (to increase op code budget):**
   It is for increasing the opcode (computational) budget which is required for internal swap calculations.

2. **AssetTransfer:**
   Sender: pool_address
   Receiver: user_address
   Index: pool_token_asset_id
   Amount: *(See formula II.B for pool token calculation)*

B. Single Asset

Transactions

1. **AssetTransfer/Pay:**
   Sender: user_address
   Receiver: pool_address
   Index: asset_id
   Amount: asset_amount

2. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“add_liquidity”, “single”, min_output]
   Foreign Assets: [pool_token_asset_id]
   Accounts: [pool_address]
   Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. issued_pool_tokens
4. asset_1_protocol_fees
5. asset_2_protocol_fees
6. asset_1_cumulative_price
7. asset_2_cumulative_price
8. cumulative_price_update_timestamp

Logs

1. input_asset_id
2. output_asset_id
3. swap_amount
4. poolers_fee_amount
5. protocol_fee_amount
6. total_fee_amount

Inner Transactions

1. **App Call (to increase op code budget):**

It is for increasing the opcode (computational) budget which is required for internal swap calculations.

2. **AssetTransfer:**
   Sender: pool_address
   Receiver: user_address
   Index: pool_token_asset_id
   Amount: *(See formula II.B for pool token calculation)*

Remove Liquidity

A. Multiple Assets Out

Transactions

1. **AssetTransfer:**
   Sender: user_address
   Receiver: pool_address
   Index: pool_token_asset_id
   Amount: pool_token_asset_amount

2. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“remove_liquidity”, min_asset_1_out, min_asset_2_out]
   Foreign Assets: [asset_1_id, asset_2_id]
   Accounts: [pool_address]
   Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. issued_pool_tokens
4. asset_1_protocol_fees
5. asset_2_protocol_fees
6. asset_1_cumulative_price
7. asset_2_cumulative_price
8. cumulative_price_update_timestamp

Inner Transactions

1. **AssetTransfer:**

Sender: pool_address
Receiver: user_address
Index: asset_1_id
Amount: *(See formula III.A)*

2. **AssetTransfer/Pay:**
   Sender: pool_address
   Receiver: user_address
   Index: asset_2_id
   Amount: *(See formula III.A)*

B. Single Asset Out

Transactions

1. **AssetTransfer:**
   Sender: user_address
   Receiver: pool_address
   Index: pool_token_asset_id
   Amount: pool_token_asset_amount

2. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“remove_liquidity”, min_asset_1_out, min_asset_2_out]
   Foreign Assets: [output_asset_id]
   Accounts: [pool_address]
   Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. issued_pool_tokens
4. asset_1_protocol_fees
5. asset_2_protocol_fees
6. asset_1_cumulative_price
7. asset_2_cumulative_price
8. cumulative_price_update_timestamp

Logs

1. input_asset_id
2. input_amount
3. swap_amount
4. output_asset_id

5. output_amount
6. poolers_fee_amount
7. protocol_fee_amount
8. total_fee_amount

Inner Transactions

1. **App Call (to increase op code budget):**
   It is for increasing the opcode (computational) budget which is required for internal swap calculations.
2. **AssetTransfer/Pay:**
   Sender: pool_address
   Receiver: user_address
   Index: output_asset_id
   Amount: *(See formula III.B)*

Swap

Transactions

1. **Input**
   a. **AssetTransfer:**
      Sender: user_address
      Receiver: pool_address
      Index: asset_1_id or asset_2_id
      Amount: input_amount

   b. **Pay:**
      Sender: user_address
      Receiver: pool_address
      Amount: input_amount

2. **AppCall:**
   a. **Mode: Fixed Input**
      Sender: user_address
      Index: tinyman_amm_v2_app_id
      OnComplete: NoOp
      App Args: [“swap”, “fixed-input”, min_output_amount]
      Foreign Assets: [asset_1_id, asset_2_id]
      Accounts: [pool_address]
      Fee: (2 * min_fee)

   b. **Mode: Fixed Output**
      Sender: user_address
      Index: tinyman_amm_v2_app_id
      OnComplete: NoOp

App Args: [“swap”, “fixed-output”, output_amount]
Foreign Assets: [asset_1_id, asset_2_id]
Accounts: [pool_address]
Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_reserves
2. asset_2_reserves
3. asset_1_protocol_fees
4. asset_2_protocol_fees
5. asset_1_cumulative_price
6. asset_2_cumulative_price
7. cumulative_price_update_timestamp

Logs

1. input_asset_id
2. input_amount
3. swap_amount
4. change
5. output_asset_id
6. output_amount
7. poolers_fee_amount
8. protocol_fee_amount
9. total_fee_amount

Inner Transactions

1. **Optional, Transfer Change, if the swap mode is *fixed-output*.**
    a. **AssetTransfer:**
    Sender: pool_address
    Receiver: user_address
    Index: input_asset_id
    Amount: *(See formula IV.B)*

    b. **Pay: (Input asset is Algo)**
    Sender: pool_address
    Receiver: user_address
    Amount: *(See formula IV.B)*

2. **Transfer Output**
    a. **AssetTransfer:**
    Sender: pool_address
    Receiver: user_address

Index: output_asset_id
Amount: *(See formula IV)*

a. **Pay: (Output asset is Algo)**
Sender: pool_address
Receiver: user_address
Amount: *(See formula IV)*

Flash Loan

Transactions

1. **AppCall:**
Sender: user_address
Index: tinyman_amm_v2_app_id
OnComplete: NoOp
App Args: [“flash_loan”, index_diff, asset_1_amount, asset_2_amount]
Foreign Assets: [asset_1_id, asset_2_id]
Accounts: [pool_address]
Fee: (3 * min_fee)

2. **AppCall** *(Group Index: index of flash loan call + index diff specified in the arguments)***:**
Sender: user_address
Index: tinyman_amm_v2_app_id
OnComplete: NoOp
App Args: [“verifiy_flash_loan”, index_diff]
Foreign Assets: []
Accounts: [pool_address]
Fee: min_fee

Side Effects

1. Flash Loan

Local State Changes

1. asset_1_cumulative_price
2. asset_2_cumulative_price
3. cumulative_price_update_timestamp

Inner Transactions

1. **AssetTransfer (If asset 1 amount is not 0):**
Sender: pool_address
Receiver: user_address
Index: asset_1_id
Amount: asset_1_amount

2. **AssetTransfer/Pay (If asset 2 amount is not 0):**
   Sender: pool_address
   Receiver: user_address
   Index: asset_2_id
   Amount: asset_2_amount

2. Verify Flash Loan

   Local State Changes

   1. asset_1_reserves
   2. asset_2_reserves
   3. asset_1_protocol_fees
   4. asset_2_protocol_fees

   Logs

   1. asset_1_output_amount
   2. asset_1_input_amount
   3. asset_1_donation_amount
   4. asset_1_poolers_fee_amount
   5. asset_1_protocol_fee_amount
   6. asset_1_total_fee_amount
   7. asset_2_output_amount
   8. asset_2_input_amount
   9. asset_2_donation_amount
   10. asset_2_poolers_fee_amount
   11. asset_2_protocol_fee_amount
   12. asset_2_total_fee_amount

Flash Swap

Transactions

1. **AppCall:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“flash_swap”, index_diff, asset_1_amount, asset_2_amount]
   Foreign Assets: [asset_1_id, asset_2_id]
   Accounts: [pool_address]
   Fee: (3 * min_fee)

2. **AppCall** *(Group Index: index of flash swap call + index diff specified in the arguments)***:**
   Sender: user_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp

App Args: [“verifiy_flash_swap”, index_diff]
Foreign Assets: [asset_1_id, asset_2_id]
Accounts: [pool_address]
Fee: min_fee

Side Effects

1. Flash Swap

Local State Changes

1. lock
2. asset_1_cumulative_price
3. asset_2_cumulative_price
4. cumulative_price_update_timestamp

Logs

To able to share data between app calls, these logs are added:

1. asset_1_balance_after_transfer
2. asset_2_balance_after_transfer

Inner Transactions

1. **AssetTransfer (If asset 1 amount is not 0):**
   Sender: pool_address
   Receiver: user_address
   Index: asset_1_id
   Amount: asset_1_amount

2. **AssetTransfer/Pay (If asset 2 amount is not 0):**
   Sender: pool_address
   Receiver: user_address
   Index: asset_2_id
   Amount: asset_2_amount

2. Verify Flash Swap

Local State Changes

1. lock
2. asset_1_reserves
3. asset_2_reserves
4. asset_1_protocol_fees
5. asset_2_protocol_fees

Logs

1. asset_1_output_amount
2. asset_1_input_amount
3. asset_1_poolers_fee_amount
4. asset_1_protocol_fee_amount

5. asset_1_total_fee_amount
6. asset_2_output_amount
7. asset_2_input_amount
8. asset_2_poolers_fee_amount
9. asset_2_protocol_fee_amount
10. asset_2_total_fee_amount

Claim Fees

Transactions

1. **AppCall:**
   Sender: any_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“claim_fees”]
   Foreign Assets: [asset_1_id, asset_2_id]
   Accounts: [pool_address, fee_collector]
   Fee: (3 * min_fee)

Side Effects

Local State Changes

1. asset_1_protocol_fees
2. asset_2_protocol_fees

Inner Transactions

1. **AssetTransfer:**
   Sender: pool_address
   Receiver: fee_collector
   Index: asset_1_id
   Amount: asset_1_protocol_fees
2.
   a. **AssetTransfer:**
      Sender: pool_address
      Receiver: fee_collector
      Index: asset_2_id
      Amount: asset_2_protocol_fees

   b. **Pay:**
      Sender: pool_address
      Receiver: fee_collector
      Amount: asset_2_protocol_fees

Claim Extra

Transactions

1. **AppCall:**
   Sender: any_address
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“claim_extra”]
   Foreign Assets: [asset__id]
   Accounts: address, fee_collector]
   Fee: (2 * min_fee)
1.

Set Fee

Transactions

1. **AppCall:**
   Sender: fee_setter
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“set_fee”, total_fee_share, protocol_fee_ratio]
   Accounts: [pool_address]
   Fee: min_fee

Side Effects

Local State Changes

1. total_fee_share
2. protocol_fee_ratio

Set Fee Collector

Transactions

1. **AppCall:**
   Sender: fee_manager
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“set_fee_collector”]
   Accounts: [new_fee_collector]
   Fee: min_fee

Side Effects

Local State Changes

1. fee_collector

Set Fee Setter

Transactions

1. **AppCall:**
   Sender: fee_manager
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“set_fee_setter”]
   Accounts: [new_fee_setter]
   Fee: min_fee

Side Effects

Local State Changes

1. fee_setter

Set Fee Manager

Transactions

1. **AppCall:**
   Sender: fee_manager
   Index: tinyman_amm_v2_app_id
   OnComplete: NoOp
   App Args: [“set_fee_manager”]
   Accounts: [new_fee_manager]
   Fee: min_fee

Side Effects

Local State Changes

1. fee_manager

State Data

Global State

1. fee_setter (bytes)
2. fee_collector (bytes)
3. fee_manager (bytes)

Local State (Pool)

1. pool_token_asset_id (uint)
2. asset_1_id (uint)
3. asset_2_id (uint)
4. total_fee_share (uint)
5. protocol_fee_ratio (uint)
6. asset_1_reserves (uint)
7. asset_2_reserves (uint)
8. issued_pool_tokens (uint)
9. asset_1_protocol_fees (uint)
10. asset_2_protocol_fees (uint)
11. lock (uint)
12. asset_1_cumulative_price (bytes)
13. asset_2_cumulative_price (bytes)
14. cumulative_price_update_timestamp (uint)

Oracle Data

The protocol computes and stores some data which can be consumed by external contracts to create TWAP price oracles for pools. The oracle related data of the pool is updated with any operation which may change the price. It is updated at most once in a block. The first operation of the pool in that block updates the oracle data before any changes are made to the pool ratio.

Pools have 3 oracle related fields in their local state:

1. asset_1_cumulative_price (bytes)
2. asset_2_cumulative_price (bytes)
3. cumulative_price_update_timestamp (uint)

Cumulative price fields are scaled by 2^64 to provide more precision. In the bootstrap operation `asset_1_cumulative_price` and `asset_2_cumulative_price` are set to zero and `cumulative_price_update_timestamp` is set to the latest block timestamp.

$$time\_delta = (latest\ block\ timestamp) - (cumulative\ price\ update\ timestamp)$$
$$asset\_1\_cumulative\_price = asset\_1\_cumulative\_price + ((asset\_2\_reserves * 2\text{^}64) * time\_delta) / ($$
$$asset\_2\_cumulative\_price = asset\_2\_cumulative\_price + ((asset\_1\_reserves * 2\text{^}64) * time\_delta) / ($$
$$cumulative\_price\_update\_timestamp = latest\ block\ timestamp$$

The cumulative price fields accumulate uint128 values represented as bytes. Even if these values accumulate until the final possible block (max uint64 value) then the total cumulative price will never exceed a uint192 or 24 bytes. For this reason these values do not overflow or wrap around.

Calculating Quotes

Calculating quotes for swaps and other operations requires knowledge of the current pool state including fee parameters, reserves, issued protocol tokens, etc. This information can be retrieved from the local state of the Tinyman app in the pool account. It is important to use fresh state data when calculating quotes as pool ratios and prices can change quickly.

Swap

Fixed Input

See Formula IV.A

Fixed Input Swap from asset 1 to asset 2

```
total_fee = (input_amount * state["total_fee_share"]) / 10000
swap_amount = input_amount - total_fee
k = state["asset_1_reserves"] * state["asset_2_reserves"]
swap_output = state["asset_2_reserves"] - (floor(k / (state["asset_1_reserves"] +
swap_amount)) + 1)
```

Allowing for a slippage of 1%, `min_output_amount` should be set as `swap_output * 0.99`

Fixed Output

See Formula IV.B

Fixed Output Swap from asset 1 to asset 2

```
k = state["asset_1_reserves"] * state["asset_2_reserves"]
swap_amount = (floor(k / (state["asset_2_reserves"] - output_amount)) + 1) -
state["asset_1_reserves"]
input_amount = floor((swap_amount * 10000) / (10000 - state["total_fee_share"]))
```

Allowing for a slippage of 1%, `input_amount` should be set as `input_amount * 1.01`

Unit Tests

Tests Sheet