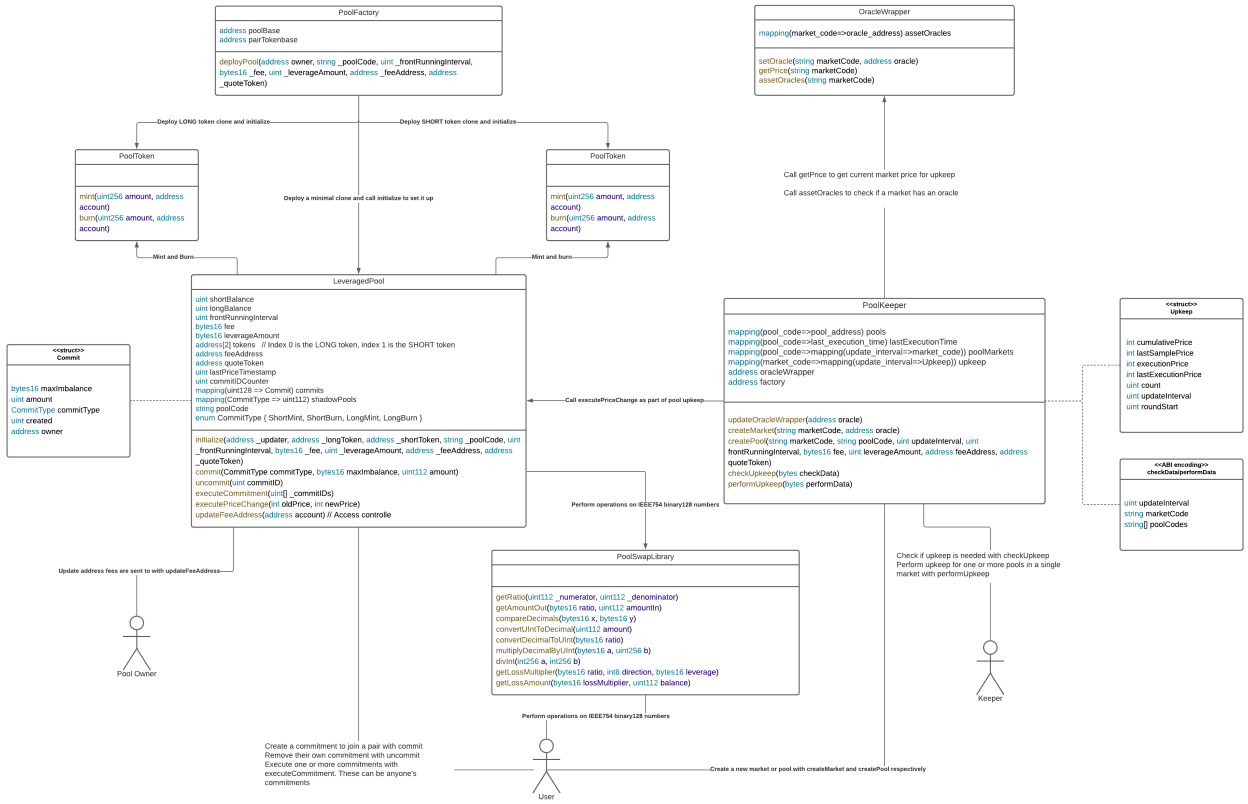
Contract Reference

Quick reference on how the contracts fit together:



PoolFactory

Used by the `PoolKeeper` to deploy a new pool and its pair tokens. The pool and the tokens are deployed as minimal clones. The factory will automatically deploy and initialize the base contracts when it is deployed.

The factory only needs to be deployed once. It can support multiple keepers and multiple pools.

Events

DeployPool

```
event DeployPool(address indexed pool, string poolCode);
```

Emitted every time a pool is deployed.

- `pool` The address of the new pool. This can be calculated deterministically using the Openzeppelin Clones library's `predictDeterministicAddress` function.
- `poolCode` The pool code for the newly deployed pool To predict the address:

```
Clones.predictDeterministicAddress(
```

```
          address(factory.poolBase()),
          keccak256(abi.encode(_poolCode)),
          address(factory)
        )
```

State changing functions

deployPool

```
  function deployPool(
      address owner,
      string memory _poolCode,
      uint32 _frontRunningInterval,
      bytes16 _fee,
      uint16 _leverageAmount,
      address _feeAddress,
      address _quoteToken
    ) external returns (address);
```

Deploys a minimal clone of the `LeveragedPool` contract and two minimal clones of an ERC20 token for the pool to use as pair tokens. The pool and tokens are initialized. The access control for the two pair tokens is set to the newly deployed pool

- `owner` The access control for the pool's `executePriceChange` function is granted for this address. Typically this will be the keeper that is requesting the deployment.

OracleWrapper

Abstracts the source of the pricing data. Currently supports chainlink compatible oracles.

Read only functions

assetOracles

```
  function assetOracles(string memory marketCode) external view returns (ad
```

Returns the oracle being used for a given market.

- `marketCode` The market to look up.

getPrice

```
  function getPrice(string memory marketCode) external view returns (int256
```

Returns the price from the chainlink oracle for the latest round.

- `marketCode` The market to look up.

State changing functions

setOracle

```
function setOracle(string memory marketCode, address oracle) external;
```

Sets the oracle address for a market. By default, this can only be used by the account that deployed the oracle wrapper.

- `marketCode` The market to set an oracle for
- `oracle` The new oracle address. Currently, this must conform to the Chain link `AggregatorV3Interface`.

PoolKeeper

Responsible for collecting average price data for an interval. Provides a single point of contact to perform price executions for multiple pools.

Events

CreatePool

```
event CreatePool(
    address indexed poolAddress,
    int256 indexed firstPrice,
    uint32 indexed updateInterval,
    string market
  );
```

Emitted when a new pool is created.

- `poolAddress` The address of the new pool
- `firstPrice` The current price of the market oracle multiplied by 1e18. The pools use pricing with a fixed point of 18 decimal places.
- `updateInterval` The number of seconds that must elapse before a pool can have a price change execution
- `market` The market code for the market the pool was created in.

CreateMarket

```
event CreateMarket(string marketCode, address oracle);
```

Emitted when a market is created.

- `marketCode` The new market's identifier
- `oracle` The oracle that the market will use for price change executions.

NewRound

```
event NewRound(
    int256 indexed oldPrice,
    int256 indexed newPrice,
    uint32 indexed updateInterval,
    string market
  );
```

Emitted when a new pricing interval occurs for a pool. This occurs when the current Unix timestamp is greater than the last price execution time plus the pool's update interval. Not all price change executions will emit this. It should only be emitted once per market/update interval pair, by the upkeep transaction that occurs first in the new interval.

- `oldPrice` The price from the penultimate price execution.
- `newPrice` The new price used for execution in the interval just passed. Both `oldPrice` and `newPrice` are an average of the price changes that have occurred during an interval (a price sample is taken whenever the price changes).
- `updateInterval` The size of the interval in seconds.
- `market` The market identifier. Price data is gathered and stored for `marketCode`/`updateInterval` pairs, to easily share data between pools on the same interval tracking the same market. These two parameters allow access to the current average price for the interval in progress.

PriceSample

```
event PriceSample(
    int256 indexed cumulativePrice,
    int256 indexed count,
    uint32 indexed updateInterval,
    string market
  );
```

Emitted when a price sample is taken for the current interval. The prices used in price change executions are the average price of a market during the interval.

- `cumulativePrice` The sum of price samples taken during the interval
- `count` The number of samples taken so far
- `updateInterval` The interval length in seconds
- `market` The market identifier for the market that's been sampled

ExecutePriceChange

```
event ExecutePriceChange(
    int256 indexed oldPrice,
    int256 indexed newPrice,
    uint32 indexed updateInterval,
    string market,
    string pool
  );
```

Emitted when a pool has a price change execution. If a pool fails to update, a `PoolUpdateError` will be emitted instead.

- `oldPrice` The price from the penultimate price execution.
- `newPrice` The new price used for execution in the interval just passed. Both `oldPrice` and `newPrice` are an average of the price changes that have occurred during an interval (a price sample is taken whenever the price changes).
- `updateInterval` The interval length in seconds
- `market` The market identifier for the market that's been sampled
- `pool` The pool that was updated

PoolUpdateError

```
event PoolUpdateError(string indexed poolCode, string reason);
```

Emitted when a pool fails a price change execution. Since pools are updated in groups, this is used to signal an issue without reverting and leaving all pools out of date.

- `poolCode` The pool's identifier
- `reason` The reason for the failure. The most likely reason is that the `ERC20.transfer` call failed when transferring the fee to the fee holder.

Read-only functions

checkUpkeep

```
function checkUpkeep(bytes calldata checkData) external view;
```

Used by pool keepers to check if an upkeep is necessary. An upkeep will occur if one of the pools requires a price execution or if the price has changed from the last sample.

- `checkData` The abi encoded data in the format `uint32, string, string[]`. In order, this is update interval, market code, and an array of pool codes.

State changing functions

updateOracleWrapper

```
function updateOracleWrapper(address oracle) external;
```

Updates the address of the oracle wrapper that the keeper uses to get price data. This can only be used by an account that has been granted the `ADMIN` role. By default, this role is granted to the account that deployed the keeper. The contract uses Openzeppelin access controls, so this can be changed post-deployment.

- `oracle` The new oracle address.

createMarket

```
function createMarket(string memory marketCode, address oracle) external;
```

Creates a new market. This must be done before a pool can be created. This only needs to be done once for each asset to be tracked. Multiple pools at different intervals can use the same oracle.

- `marketCode` The unique market identifier. There is no restriction on format, but it must not already exist in the keeper.
- `oracle` The oracle to use for the market.

createPool

```
function createPool(
    string memory marketCode,
    string memory poolCode,
    uint32 updateInterval,
    uint32 frontRunningInterval,
    bytes16 fee,
    uint16 leverageAmount,
    address feeAddress,
    address quoteToken
  ) external;
```

Creates a pool in a given market.

- `marketCode` The identifier for the market to create the pool in. This must exist before the pool is created.
- `poolCode` The name of the pool. There are no restrictions on the format, however, the pool must not already exist in the keeper.
- `updateInterval` The frequency in seconds that the pool will be updated
- `frontRunningInterval` The amount of time a user must commit to adding or withdrawing to the pool before the movement of funds can occur. This must be smaller than the update interval.
- `fee` The percentage fee to be charged with every price change execution. This is per update interval, so for a 2% annual fee at daily updates, the fee would be 0.02 / 52 = ~0.00038. To generate the number in the correct format, the `PoolSwapLibrary.getRatio` method

should be used.
- `leverageAmount` The amount to be applied in the power leverage calculation. Internally this is stored as a decimal so the `PoolSwapLibrary.convertDecimalToUInt` method should be used when retrieving a pool's leverage for users to view.
- `feeAddress` The address to send fees to
- `quoteToken` The address of the digital asset that the pool is demarcated in

performUpkeep

```
function performUpkeep(bytes calldata performData) external;
```

Performs upkeep on one or more pools. This will either take a price sample (and optionally execute a price change for the pools), or it will start a new interval and execute a price change for the pools in the calldata.

- `performData` The abi encoded data in the format `uint32, string, string[]`. In order, this is update interval, market code, and an array of pool codes.

LeveragedPool

Manages the short and long pairs of a pool. This is the contract most users will be interacting with as they commit to deposit and withdraw.

Events

PoolInitialized

```
event PoolInitialized(
    address indexed longToken,
    address indexed shortToken,
    address quoteToken,
    string poolCode
  );
```

Emitted when a newly deployed pool is initialized. A pool that isn't initialized in the same transaction as it was deployed should be considered suspect (there should be a `DeployPool` event in the same transaction). A pool that hasn't emitted this event should not be used as it is vulnerable to being taken over by an attacker.

- `longToken` The address of the token that represents the long pool
- `shortToken` The address of the token that represents the short pool
- `quoteToken` The digital asset in the pool
- `poolCode` The pool identifier

CreateCommit

```
event CreateCommit(
  uint128 indexed commitID,
  uint128 indexed amount,
  bytes16 indexed maxImbalance,
  CommitType commitType
);
```

Emitted when a commit is created. This forms the user's record of commits, as the commit details are not retrievable without the commit ID.

- `commitID` The ID of the commit, to be used when withdrawing or executing the commit.
- `amount` The amount that was committed
- `maxImbalance` The difference between the pools that the user-specified. If the commit is executed and this is smaller than the resulting difference, the transaction will revert.
- `commitType` The type of commit (long burn, short burn, long mint, short mint)

RemoveCommit

```
event RemoveCommit(
    uint128 indexed commitID,
    uint128 indexed amount,
    CommitType indexed commitType
  );
```

Emitted when a commit is withdrawn.

- `commitID` The ID of the withdrawn commit
- `amount` The number of tokens that were returned to the user
- `commitType` The type of commit that was withdrawn

ExecuteCommit

```
event ExecuteCommit(uint128 commitID);
```

Emitted when a commit is executed. Commit execution is the transfer of funds from the pool's shadow balance into the live balances.

PriceChange

```
event PriceChange(
    int256 indexed startPrice,
    int256 indexed endPrice,
    uint112 indexed transferAmount
  );
```

Emitted when a price change execution occurs.

- `startPrice` The price from the last execution interval
- `endPrice` The price from the current execution interval

State changing functions

initialize

```
function initialize(
  address _updater,
  address _longToken,
  address _shortToken,
  string memory _poolCode,
  uint32 _frontRunningInterval,
  bytes16 _fee,
  uint16 _leverageAmount,
  address _feeAddress,
  address _quoteToken
) external;
```

Initializes a minimal clone of a pool contract. This can only be run once. Ordinarily, this will be executed by the pool factory in the same transaction as the deployment.

commit

```
function commit(
    CommitType commitType,
    bytes16 maxImbalance,
    uint112 amount
  ) external;
```

Used to create a commitment to add or remove funds from one of the pool's pairs.

- `maxImbalance` The maximum difference between the pools that the user will tolerate. This is the ratio between the long and the short live balances. The value should be generated using the `PoolSwapLibrary.getRatio` method.

uncommit

```
function uncommit(uint128 commitID) external;
```

Used to withdraw a commitment. This cannot be used to withdraw a commit that the user doesn't own.

executeCommitment

```
function executeCommitment(uint128[] memory _commitIDs) external;
```

Used to execute the transfer of funds from the shadow pool balance into the live pool balance. A user can execute commits that they do not own.

executePriceChange

```
function executePriceChange(int256 oldPrice, int256 newPrice) external;
```

Used by the pool keeper to move funds between the long and short pool balances based on the change in the price of the underlying market asset. This function is only able to be called by the pool keeper (or another party, as specified during deployment of the pool).

updateFeeAddress

```
function updateFeeAddress(address account) external;
```

Used by the pool owner to change the address that the pool fees are transferred to every price execution. This address is the only one allowed to change the fee address. Care must be taken to not change it to an address outside of the pool owner's control.

PoolSwapLibrary

Utilizes the ABDKMathQuad library to work with 128-bit floating-point numbers (IEEE754 binary128). Provides utility functions to support the `LeveragedPool` contract, and any potential frontend clients.

Read only functions

getRatio

```
function getRatio(uint112 _numerator, uint112 _denominator)
    external
    pure
    returns (bytes16);
```

Calculates `_numerator/_denominator` and returns the result as an IEEE754 binary128 number.

getAmountOut

```
function getAmountOut(bytes16 ratio, uint112 amountIn)
    external
    pure
    returns (uint112)
```

Returns the amount of `amountIn` to transfer based on the `ratio`.

compareDecimals

```
function compareDecimals(bytes16 x, bytes16 y) external pure returns (int
8);
```

Compares two IEEE754 binary128 numbers and returns:

- `-1` if the first is lower than the second
- `0` if they are the same value
- `1` if the first number is larger than the second

convertUIntToDecimal

```
function convertUIntToDecimal(uint112 amount)
    external
    pure
    returns (bytes16)
```

Converts a `uint` value to an IEEE754 binary128 number.

convertDecimalToUInt

```
function convertDecimalToUInt(bytes16 ratio) external pure returns (uint2
6);
```

Converts an IEEE754 binary128 number to a `uint256`.

multiplyDecimalByUInt

```
function multiplyDecimalByUInt(bytes16 a, uint256 b)
    external
    pure
    returns (bytes16)
```

Returns the product of an IEEE754 binary128 number and a `uint256` as an IEEE754 binary128 number.

divInt

```
function divInt(int256 a, int256 b) external pure returns (bytes1
6);
```

Divides two `int256` values and returns the result as an IEEE754 binary128 value.

getLossMultiplier

```
function getLossMultiplier(
    bytes16 ratio,
    int8 direction,
    bytes16 leverage
  ) external pure returns (bytes16)
```

Calculates the multiplier to apply to the balance of the losing pool. The formula it uses is:

```
Ratio R = old price / new price
Loss multiplier LM = (R < 1 ? 1 : 0) * R + (R >= 1 ? 1 : 0) * 1 / R
Adjusted loss multiplier = 1 - LM ^ leverage
```

The exact implementation for the power function uses the following formula for performance reasons.

$$x^y = 2^{y \cdot \log_2 x}$$

getLossAmount

```
function getLossAmount(bytes16 lossMultiplier, uint112 balance)
    external
    pure
    returns (uint256);
```

Applies a loss multiplier to an amount and returns the amount that should be transferred.

one and zero

Getter functions for pre-generated IEEE754 binary128 values for 1 and 0 respectively.

PoolToken

A standard ERC20 contract with mint and burn methods under the pool's control. The contract is compatible with the minimal clone proxy type.