Kine: The Liquidity Pool Protocol

Abstract

Kine is a decentralized protocol which establishes general purpose liquidity pools backed by a customizable portfolio of digital assets. The liquidity pool allows traders to open and close derivatives positions according to trusted price feeds, avoiding the need of counterparties. Kine lifts the restriction on existing peer-to-pool (aka peer-to-contract) trading protocols, by expanding the collateral space to any Ethereum-based assets and allowing third-party liquidation.

1 Introduction

1.1 CFD and Synthetic Financing

One of the services traditionally provided by prime brokers to hedge funds is the provision of leverage, that is, loans extended to hedge funds to conduct trading and enhance returns. In markets with enhanced capital requirements, prime brokers have begun to offer an alternative means of providing hedge fund clients with leveraged exposure to securities. Known as synthetic financing, this alternative requires the prime broker to enter into derivatives contracts with the clients to replicate the desired exposure.

A contract for difference (CFD) is a contract between two parties, stipulating that the buyers (long position) will pay to the seller (short position) the difference between the current value of an asset and its value at contract time. CFDs were initially used by hedge funds and institutional traders to gain exposures to certain markets. In the late 1990s, CFDs were introduced to retail traders and popularised by innovative online trading platforms.

1.2 Peer-to-Pool Trading

In traditional finance, derivatives are normally traded as peer-to-peer bilateral contracts. Several decentralised finance protocols such as Synthetix, Hegic and FinNexus move derivatives trading to peer-to-pool models. Liquidity is collected together in the collateral pools. The pools are the counterparties to all net positions, while providing collaterals to them. Risks and trading fees are shared across the entire group of liquidity providers.

Orders submitted to a peer-to-pool engine are filled at an exchange rate through price feeds supplied by an oracle. This provides infinite liquidity up to the total value of the pool and zero slippage.

1.3 Multi Collateral Debt Pool

The stake in a liquidity pool is represented by a collateralized debt position, whose price is determined by the pool's profit-and-loss from market making activities. The outstanding debt has to be repaid before a liquidity provider (aka staker) can withdraw pledged collaterals. In return, the stakers receive fees generated by the trading platform.

Liquidity pools backed by a diverse portfolio of digital assets will prevail single-asset pools, as the former allows customised combination of collaterals that improves the capital efficiency for staking. Moreover, such pools are less likely to fall in death spirals as diversified collateral portfolios are more resistant to market turmoils.

Risky-asset-backed stablecoins, such as Dai (by MakerDAO) and sUSD (by Synthetix) are essentially collateralised lending systems. Dai does not have an endogenous liquidity pool and the borrowers (i.e. vault owners) only need to pay interests in the form of stability fees. Synthetix pioneered in debt-based liquidity pool where sUSD minters' debt value is subject to a varying debt price.

Compound is a more versatile lending system, as it accepts portfolio collateral from an expandable set of digital assets, and allows third-party arbitrageurs to provide liquidity in extreme market conditions.

2 The Kine Protocol

Kine is a decentralized protocol which establishes general purpose liquidity pools backed by a customised portfolio of digital assets. At its core, the Kine Protocol is a collateralised lending system. While the collaterals are general ERC-20

assets, the lending asset is a special purpose token representing a stake in a liquidity pool.

2.1 Staking Assets

Assets staked into the contracts increase the user's debt limit. The debt limit equals staking asset's market value times Collateral Factor. Staking assets' value is governed by price feed provided by on-chain oracles. Collateral Factor is a system-wide parameter determined by the asset's price volatility and liquidity.

Users have the flexibility of choosing one or more supported staking assets and form their own collateral portfolio. They can increase or reduce their collaterals as long as there are enough unused debt limit.

2.2 Minting kUSD and Debts

Users with unused debt limit can mint kUSD, a synthetic USD-pegging digital asset backed by the liquidity pool. kUSD is the only asset accepted by Kine Exchange, a peer-to-pool derivatives trading platform providing multi-asset exposure with zero-slippage trading experience.

Users incur a Multi Collateral Debt (MCD) when they mint kUSD. The MCD value can increase or decrease independent of their original minted value, based on the net exposures taken by the liquidity pool. The pool provides liquidity to all trading pairs quoted on Kine Exchange. It accumulates and distributes fees and fundings to the stakers.

Stakers act as a pooled counterparty to all traders on Kine Exchange and take on the risk of MCD price movement. When the liquidity pool incurs a trading loss, the MCD price rises and the debt values of all stakers increase proportionally; alternatively when the liquidity pool posts a trading profit, the MCD price falls and debt values decrease. Thebu stakers have the option of hedging this risk by taking positions external to Kine Exchange.

2.3 Income Distribution

Trading fees and fundings collected by Kine Exchange will be distributed to the liquidity pool's stakers. The exchange will accumulate incomes in kUSD, and convert it into KINE tokens through 3rd-party DEX such as Uniswap.

Stakers can periodically claim their rewards from the distribution contract.

2.4 Burning kUSD

When a staker wants to withdraw collaterals or exit the system, they must pay down their debt by burning kUSD. If the MCD pool fluctuates while they are staked, they may need to burn more or less kUSD than they originally minted. The process of reducing debt is as follows:

1. Certain amount of kUSD is converted to MCD by the latest debt price supplied by the oracle.
2. The MCD are returned to the lending system and the unused debt limit increases.
3. With unused debt limit, user may withdraw part or all of staked assets.

2.5 Liquidation

If a staker's outstanding MCD value exceeds its debt limit, a portion of the outstanding MCD may be repaid by burning kUSD in exchange of an amplified amount of the staker's collateral. This incentives arbitrageurs to swiftly step in to reduce the staker's exposure, and eliminate the protocol's risk.

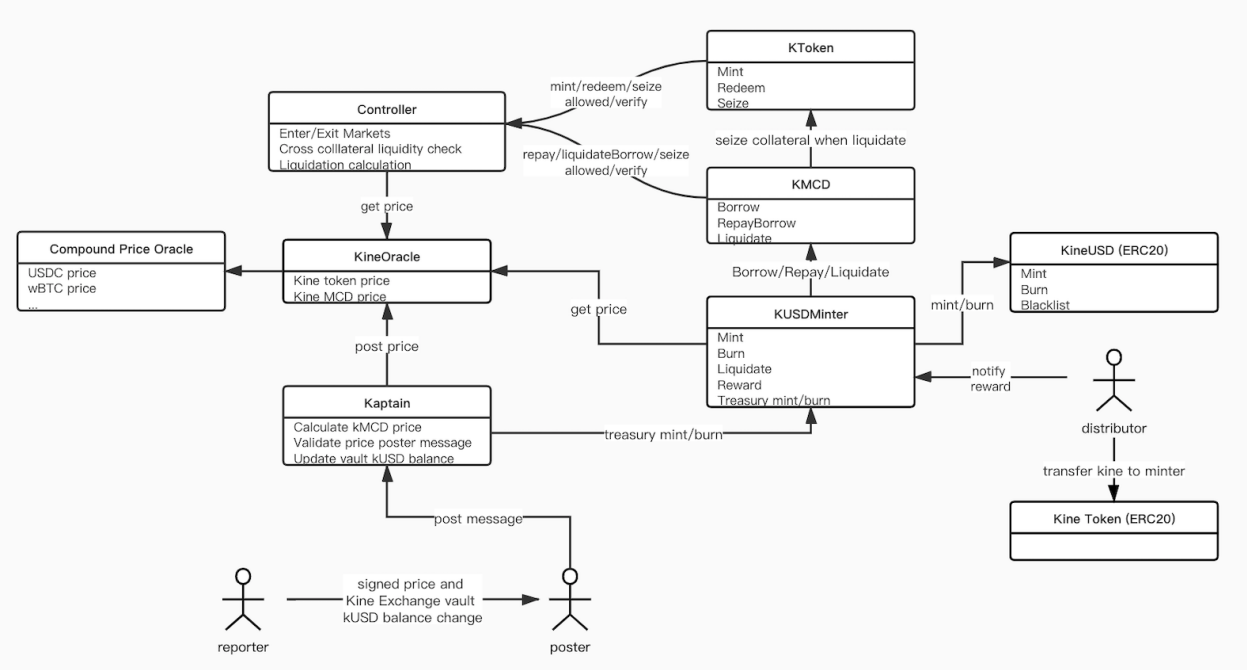
The proportion eligible to be closed, a close factor, is the portion of the MCD that can be repaid, and ranges from 0 to 1, such as 25%. The liquidation process may continue to be called until the user’s outstanding MCD is less than their debt limit. Any Ethereum address that possesses kUSD may invoke the liquidation function.

3 Implementation & Architecture

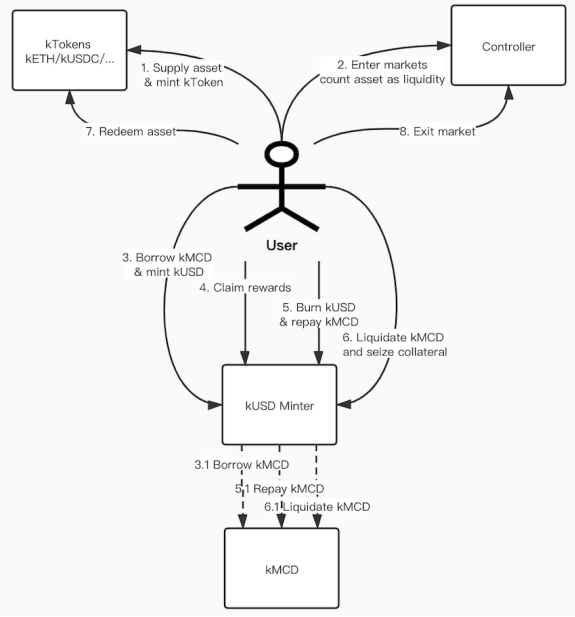
This section lays out the detailed design and implementation of the Kine Protocol. `KToken` contracts accept staking assets from users and mint transferrable kTokens from which users can redeem the staking assets. `Controller` contract acts as the risk controller by allowing or rejecting user actions according to their collateral vs. liquidity states. `KMCD` contracts keep

records of Multi Collateral Debts (MCD) and work with `Minter` contracts to control to supply of kUSD. `Kaptain` contract along with `KineOracle` contract keep updating asset and debt prices to feed `Controller` and `Minter` for their calculations.

The architecture of contracts is as below.



Users may interact with `KToken`, `Controller` and `Minter` for majority of Kine Protocol functionalities.



3.1 KToken Contracts

`KToken` contracts implement the functionality of staking assets as collateral to users. By staking assets to `KToken` contracts, stakers gain (mint) kTokens representing their staking balances and may use these balances as liquidation to increase their Liquidity Debt Limit in order to mint kCurrency. The minted kTokens can be redeemed partially or in full to withdraw the

assets, or be transferred to others following ERC20 protocol. Before kTokens being redeemed or transferred, `Controller` contract will check if the owner's collateral is sufficient against its Liquidation Debt to allow or reject transactions.

3.1.1 Contracts Structure

Each type of staking asset is contained by a `KToken` contract. There are currently two kinds of kTokens, `KEther` and `KErc20`, which manage ether and ERC20 tokens respectively. `KErc20` follows DelegateProxy pattern for upgradability, where `KErc20Delegator` is the storage contract while `KErc20Delegate` is the logic implementation contract.

3.1.2 Functions

Mint

Transfer underlying ERC20 tokens into `KErc20` contract or ether to `KEther` contract as collateral and mint equivalent amount of kTokens to `msg.sender`.

`KErc20`

```
function mint(uint mintAmount) external returns (uint)
```

`KEther`

```
function mint() external payable returns (uint)
```

Redeem

Burn kTokens from `msg.sender`, and transfer equivalent amount of ERC20 tokens from `KErc20` contract or ether from `KEther` contract to msg.sender. Will check `msg.sender`'s liquidity first through `Controller` contract and be rejected if liquidity is not sufficient after transfer.

```
function redeem(uint redeemTokens) external
```

Transfer

Transfer kTokens from `msg.sender` to another address. Will check `msg.sender`'s liquidity first through `Controller` contract and be rejected if liquidity is not sufficient after transfer.

```
function transfer(address dst, uint256 amount) external returns (bool)
```

3.1.3 Key Events

```
//Emitted upon a successful Mint.
Mint(address minter, uint mintAmount, uint mintTokens)
//Emitted upon a successful Redeem.
Redeem(address redeemer, uint redeemTokens)
//Emitted upon a successful Transfer.
Transfer(address indexed from, address indexed to, uint amount)
```

3.2 KMCD Contracts

Multi Collateral Debt are implemented by `KMCD` contracts. Each `KMCD` contract can hold its own logic on debt creation and utilization.

Currently there are only one `KMCD` contract in Kine Protocol, which incur debt when user mint kUSD through `KUSDMinter` contract. The minted kUSD shall be used to trade synthetic assets in Kine Exchange.

3.2.1 Contracts Structure

The `KMCD` contract works as a debt ledger and cannot interact directly with external users. When users mint/burn kUSD on `KUSDMinter` contract, the minter will call `KMCD` contract to borrow/repay debt on their behalf.

`KMCD` also follows DelegateProxy pattern for upgradability.

3.2.2 Functions

BorrowBehalf

Only called by `KUSDMinter` contract when user mint kUSD. Will create a kMCD borrow balance to the user. The amount borrowed must be less than the borrower's liquidity.

```
function borrowBehalf(address payable borrower, uint borrowAmount) onlyMinter external
```

RepayBorrowBehalf

Only called by `KUSDMinter` contract when user burn kUSD. Will reduce the user's kMCD borrow balance.

```
function repayBorrowBehalf(address borrower, uint repayAmount) onlyMinter external
```

LiquidateBorrowBehalf

Only called by `KUSDMinter` contract when a liquidator burns kUSD to liquidate another borrower. When a user becomes under-collateralized (has negative liquidity), a third party can liquidate its kMCD by repaying some or all (`Close Factor`) of its borrowed kMCD and receive an amplified (`Liquidation Incentive`) amount of collateral.

```
function liquidateBorrowBehalf(address liquidator, address borrower, uint repayAmount, KTokenInterface kTokenCollateral) onlyMinter externa
```

BorrowBalance

Returns given account's borrowed kMCD balance.

```
function borrowBalance(address account) public view returns (uint)
```

TotalBorrows

Returns total amount of borrowed kMCD.

```
function totalBorrows() public view returns (uint)
```

3.2.3 Key Events

```
//Emitted upon a successful Borrow.
Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows)
//Emitted upon a successful RepayBorrow.
RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint totalBorrows)
//Emitted upon a successful LiquidateBorrow.
LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address kTokenCollateral, uint seizeTokens)
```

3.3 kCurrency and Minters

Each MCD is designed to mint one kCurrency through its corresponding minter. Currently there is only one kCurrency - `KineUSD` (kUSD) and its minter `KUSDMinter`.

`KineUSD` is an ERC20 which can be paused and prevent blacklist users to transfer. `KUSDMinter` is the minter of `KineUSD` and the interface to users to transit their debt to kUSD or vice versa.

`KUSDMinter` also distributes rewards periodically to stakers. The rewards are added periodically by reward distributor according to Kine Tokenomics and Kine Exchange trading fees. The accrued rewards can be claimed following a vesting algorithm.

3.3.1 Contract Structure

`KineUSD` only allow the `KUSDMinter` to mint/burn kUSD. `KUSDMinter` call `KMCD` to borrow/repay debt on behalf of users which actions will go through `Controller` for risk control. `KUSDMinter` also utilize accrued reward mechanism to calculate users' rewards status.

`KUSDMinter` follows DelegateProxy pattern for upgradablity.

3.3.2 Functions

`KineUSD`

Mint

Only called by `KUSDMinter`, mint kUSD to account.

```
function mint(address account, uint amount) external onlyMinter
```

Burn

Only called by `KUSDMinter`, burn kUSD from account.

```
function burn(address account, uint amount) external onlyMinter
```

`KUSDMinter`

Mint

Borrow kMCD on behalf of user according to kMCD price and mint specified amount of kUSD to `msg.sender`. Borrowing kMCD will trigger `Controller` to check user's liquidity to allow or reject the transaction. There is a start time only after which can users mint kUSD.

```
function mint(uint kUSDAmount) external checkStart updateReward(msg.sender)
```

Burn

Burn specified amount of kUSD from `msg.sender` and repay kMCD according to kMCD price on behalf of user. There is a start time only after which can users burn kUSD. There is a cool-down time before user can burn kUSD after last kUSD minting. This is to prevent bots from front running MCD price update by taking and repaying debt in short periods.

```
function burn(uint kUSDAmount) external checkStart afterCooldown(msg.sender) updateReward(msg.sender)
```

BurnMax

Burn equivalent kUSD to `msg.sender`'s outstanding kMCD debt value. If kUSD balance is not sufficient, will burn all kUSD to repay debt. There is a start time only after when can users burn kUSD. There is a cool-down time before user can burn kUSD after last kUSD minting. This is to prevent bots from front running MCD price update by taking and repaying debt in short periods.

```
function burnMax() external checkStart afterCooldown(msg.sender) updateReward(msg.sender)
```

Liquidate

Burn `msg.sender`'s kUSD and call `KMCD` `liquidateBehalf` function (see `KMCD`) on behalf of `msg.sender` to liquidate staker's kMCD. If kUSD amount need to be burnt reached given `maxBurnKUSDAmount`, will revert.

```
function liquidate(address staker, uint unstakeKMCDAmount, uint maxBurnKUSDAmount, address kTokenCollateral) external checkStart updateRewa
```

Earned

Return account's accrued reward.

```
function earned(address account) public view returns (uint)
```

Claimable

Return account's matured reward. User's accrued reward will mature gradually in a given release period. The proportion of matured reward to accrued reward is the proportion of past time since last claim to the release period.

```
function claimable(address account) external view returns (uint)
```

GetReward

Transfer matured reward to `msg.sender` and start a new release period for left and new accrued reward.

```
function getReward() external checkStart updateReward(msg.sender)
```

TreasuryMint

Only called by Kine Exchange treasury account (see `Kaptain`), mint kUSD to Kine vault to keep kUSD total supply in line with the total debt value of the pool.

```
function treasuryMint(uint amount) external onlyTreasury
```

TreasuryBurn

Only called by Kine Exchange treasury account (see `Kaptain`), burn kUSD from Kine vault to keep kUSD total supply in line with the total debt value of the pool.

```
function treasuryBurn(uint amount) external onlyTreasury
```

3.3.3 Key Events

`KineUSD`

```
//Emitted upon a successful Transfer/Mint/Burn
Transfer(address indexed from, address indexed to, uint256 value)
```

`KUSDMinter`

```
//Emitted upon a successful Mint.
Mint(address indexed user, uint mintKUSDAmount, uint stakedKMCDAmount, uint userStakesNew, uint totalStakesNew)
//Emitted upon a successful Mint.
Burn(address indexed user, uint burntKUSDAmount, uint unstakedKMCDAmount, uint userStakesNew, uint totalStakesNew)
//Emitted upon a successful BurnMax.
BurnMax(address indexed user, uint burntKUSDAmount, uint unstakedKMCDAmount, uint userStakesNew, uint totalStakesNew)
//Emitted upon a successful Liquidate.
Liquidate(address indexed liquidator, address indexed staker, uint burntKUSDAmount, uint unstakedKMCDAmount, uint stakerStakesNew, uint tot
//Emitted upon a successful TreasuryMint.
TreasuryMint(uint amount)
```

```
//Emitted upon a successful TreasuryBurn.
TreasuryBurn(uint amount)
```

3.3.4 Reward Release

User's accrued rewards in `KUSDMinter` will gradually mature in a release period. Every time user claim rewards, the release timer will be updated. The matured reward of total accrued rewards is calculated as :

$$Reward_{matured} = min(1, \frac{Time_{curernt} - Time_{lastClaim}}{ReleasePeriod}) \cdot Reward_{accrued}$$

Implementation in code as:

```
function claimable(address account) external view returns (uint) {
      uint accountNewAccruedReward = earned(account);
      uint pastTime = block.timestamp.sub(accountRewardDetails[account].lastClaimTime);
      uint maturedReward = accountNewAccruedReward.mul(1e18).div(rewardReleasePeriod).mul(pastTime).div(1e18);
      if (maturedReward > accountNewAccruedReward) {
          maturedReward = accountNewAccruedReward;
      }
      return maturedReward;
  }
```

3.3.5 Liquidation

When user's kMCD debt exceeds its liquidity due to decrease in collateral value or increase in kMCD price, other users can liquidate the borrower's kMCD by repaying its debt, and seize an amplified (`Liquidation Incentive`) amount of the borrower's collateral. Calculation of account liquidity can be found in 3.5.4.

3.3.6 Burn Cooldown

Arbitrageurs may monitor price feed transactions and front-run the oracle by minting kUSD prior to a falling MCD price update, and repaying debt with less kUSD right after the price update. The arbitrage action hurts stakers' profit since they drain the profits and enlarge the loss of stakers.

We introduce the Burn Cooldown mechanism to prevent kUSD burning right after minting. Users have to wait a cooldown time before they can burn every time after they mint. And the cooldown time shall be set larger than one interval time of price post cycles. So if arbitrageurs front-run one price feed to mint kUSD, they have to wait until the next price feed and they can't not be sure the next MCD price still benefit them.

3.3.7 Treasury & Vault

When minted kUSD transferred to Kine Exchange for trading synthetic assets, the Multi Collateral Debt Pool value begin to vary (MCD price begin to vary). Since trades happens off chain, Kine Exchange is responsible to report debt pool value periodically back on chain to update MCD price. We treat the kUSD transferred to Kine Exchange as transferred to a vault account, the vary of debt pool value can just be reflected into the vault kUSD balance change then.

`KUSDMinter` stores a vault account address and only allows treasury to update vault's kUSD balance to keep debt pool value updated on chain. And the treasury is `Kaptain` which drives the updates of vault kUSD balance change together with kMCD and Kine token prices, see 3.4 Price Feeds. Calculation of vault kUSD balance change happens in Kine Exchange and follows below algorithm:

$$kUSD^{t}_{delta} = PnL^{t-1}_{pool} - PnL^{t}_{pool}$$

$$kUSD^{t+}_{supply} = kUSD^{t-}_{supply} + kUSD^{t}_{delta}$$

- $kUSD^{t}_{delta}$: Vault's kUSD balance change at time t . To be calculated and posted by Kine Exchange through `Kaptain`
- PnL^{t}_{pool}: Profit and loss of the liquidity pool on Kine Exchange at time t
- PnL^{t-1}_{pool}: Profit and loss of the liquidity pool on Kine Exchange at previous update $t-1$

- $kUSD^{t+}_{supply}$: Total supply of kUSD immediately after time t
- $kUSD^{t-}_{supply}$: Total supply of kUSD immediately before time t

3.4 Price Feeds

Kine Protocol adopts the `OpenOraclePriceData` contracts by Compound as price feeds for common assets. There is a customised view contract `KineOracle` to store prices that are signed by reporters hosted by Kine.

The MCD price is defined to be the product of the total supply of kUSD $kUSD_{supply}$ divided by the total borrowed balance of MCD $kMCD_{balance}$:

$$Price_{mcd} = \frac{kUSD_{supply}}{kMCD_{balance}}$$

kMCD price is calculated immediately after Vault kUSD balance getting updated. `Kaptain` contract is responsible for updating kUSD balance in Kine vault via `KUSDMinter`, calculating kMCD price and posting it along with other token prices to `KineOracle` all in the same transaction.

Kine vault kUSD balance change is calculated from Kine Exchange, so only Kine Exchange can post the prices for now.

3.4.1 Contracts Structure

`KineOracle` only accept prices posted from `Kaptain` contract.

`Kaptain` contract only accept Kine Exchange poster call, and will validate if posted message is signed by Kine reporter.

3.4.2 Functions

`KineOracle`

PostPrices

Post an array of prices to `OpenOraclePriceData` and saves them to `KineOracle`. Prices are expected to be in USD with 6 decimals of precision.

```
function postPrices(bytes[] calldata messages, bytes[] calldata signatures, string[] calldata symbols) external onlyKaptain
```

PostMcdPrice

Save kMCD price to `KineOracle`. Price is expected to be in USD with 6 decimals of precision.

```
function postMcdPrice(uint mcdPrice) external onlyKaptain
```

Price

Get the most recent price for a token in USD with 6 decimals of precision.

```
function price(string memory symbol) external view returns (uint)
```

GetUnderlyingPrice

Get the most recent price for an underlying token of kToken in USD scaled by $10^{(36-tokenDecimals)}$.

```
function getUnderlyingPrice(address kToken) external view returns (uint)
```

`Kaptain`

Steer

Called only by Kine Exchange poster, will call `KUSDMinter`'s `treasuryMint`/`treasuryBurn` to update Kine Exchange vault kUSD balance, calculate kMCD price and post prices to `KineOracle` in the same transaction.

```
function steer(bytes calldata message, bytes calldata signature) external onlyPoster
```

3.4.3 Key Events

`KineOracle`

```
//Emitted upon a successful Price Update
PriceUpdated(string symbol, uint price)
```

`Kaptain`

```
//Emitted upon a successful Steer
Steer(uint256 scaledMCDPrice, bool isVaultIncreased, uint256 vaultKusdDelta, uint256 reporterNonce)
```

3.5 Controller

`Controller` implements the risk management functionality of the Kine Protocol. `Controller` evaluates users' liquidity on their attempts to withdraw collaterals or take on more debts. It projects user-selected (by calling `Enter Markets` and `Exit Market`) staking assets in `KToken` contracts by `Collateral Factor` and asset prices (through `KineOracle`) to user's liquidity, and control the amount of debt user can incur (`borrow`) or reduce (`repay`) in consequence. When a user becomes under-collateralized, `Controller` calculates the proportion of its debt subject to liquidation by `Close Factor`, and the amplified amount of collaterals by `Liquidation Incentive` to be seized by liquidators.

3.5.1 Contracts Structure

`Controller` holds references of all supported `KToken`/`KMCD` instances, `KineOracle` instance and provides functions to evaluate if transactions in `KToken`, `KMCD` are allowed or not. In certain conditions, `admin` and `pauseGuardian` can `pause` and `unpause` through `Controller` contract.

`Controller` follows DelegateProxy pattern for upgradablity.

3.5.2 Functions

EnterMarkets

Enter markets, mark a list of staked `KToken` assets as account's liquidity providers.

```
function enterMarkets(address[] memory kTokens) public
```

ExitMarket

Exit market, remove a `KToken` asset from sender's account liquidity calculation.

```
function exitMarket(address kTokenAddress) external
```

GetAssetsIn

List markets the account is currently entered into.

```
function getAssetsIn(address account) external view returns (KToken[] memory)
```

Markets

List all markets with each market's `Collateral Factor` that `Controller` currently support.

```
function markets(address kTokenAddress) view returns (bool, uint)
```

GetAccountLiquidity

Get account's liquidity.

```
function getAccountLiquidity(address account) public view returns (uint, uint)
```

CloseFactor

Get the percentage of a liquidate-able account's kMCD borrow that can be repaid by liquidator in a single liquidate action.

```
function closeFactorMantissa() view returns (uint)
```

LiquidationIncentive

Get the amplify factor of seizing collateral in liquidation.

```
function liquidationIncentiveMantissa() view returns (uint)
```

3.5.3 Key Events

```
//Emitted upon a successful Enter Market.
MarketEntered(KToken kToken, address account)
//Emitted upon a successful Exit Market.
MarketExited(KToken kToken, address account)
```

3.5.4 Liquidity Calculation

When a user transfers/redeems kTokens, borrows kMCD to mint kUSD, exits kToken markets or becomes the target of liquidation, `Controller` calculates the account's liquidity first to allow or reject the transaction. The calculation of account liquidity is as below:

$$L = \sum_{i=1}^{N} Ba_i \cdot AP_i \cdot CF_i - \sum_{j=1}^{M} Bd_j \cdot DP_j$$

- L: Liquidity
- N: Number of kToken markets account has entered in
- Ba_i: Balance of i_{th} kToken account owns
- AP_i: Price of i_{th} kToken underlying asset
- CF_i: Collateral Factor of i_{th} kToken
- M: Number of MCD pools account has participated (currently M=1
- Bd_j: Borrow balance of j_{th} MCD account borrowed, currently Kine only have one MCD
- DP_j: Price of j_{th} MCD

Calculation in codes is as below:

```
// For each asset the account is in
        KToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            KToken asset = assets[i];

            // Read the balances from the kToken
            (vars.kTokenBalance, vars.borrowBalance) = asset.getAccountSnapshot(account);
            vars.collateralFactor = Exp({mantissa : markets[address(asset)].collateralFactorMantissa});

            // Get the normalized price of the asset
            vars.oraclePriceMantissa = oracle.getUnderlyingPrice(address(asset));
            require(vars.oraclePriceMantissa != 0, "price error");
            vars.oraclePrice = Exp({mantissa : vars.oraclePriceMantissa});

            // Pre-compute a conversion factor
            vars.tokensToDenom = mulExp(vars.collateralFactor, vars.oraclePrice);

            // sumCollateral += tokensToDenom * kTokenBalance
            vars.sumCollateral = mulScalarTruncateAddUInt(vars.tokensToDenom, vars.kTokenBalance, vars.sumCollateral);

            // sumBorrowPlusEffects += oraclePrice * borrowBalance
            vars.sumBorrowPlusEffects = mulScalarTruncateAddUInt(vars.oraclePrice, vars.borrowBalance, vars.sumBorrowPlusEffects);

            // Calculate effects of interacting with kTokenModify
            if (asset == kTokenModify) {
                // redeem effect
                // sumBorrowPlusEffects += tokensToDenom * redeemTokens
                vars.sumBorrowPlusEffects = mulScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.sumBorrowPlusEffects);

                // borrow effect
                // sumBorrowPlusEffects += oraclePrice * borrowAmount
                vars.sumBorrowPlusEffects = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
            }
        }

        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (0, vars.sumBorrowPlusEffects - vars.sumCollateral);
        }
```

`Controller` will iterate all kToken markets that user has entered in, sum all kToken balances multiplied by its asset price as total liquidity, sum all kMCD borrows multiplied by its debt price as total debt, count in additional liquidity or debt effect caused by user action, to conclude the final liquidity and shortfall (if liquidity is negative will present in shortfall as a positive number).