GMX is essentially a perpetual contract exchange where traders and liquidity providers (LPs) bet against each other. When users profit, LPs lose; when users lose, LPs profit. GMX users are mainly divided into two categories: traders and liquidity providers. The primary actions for traders include opening, closing, and swapping positions, while LPs mainly engage in buying and selling GLP to provide or remove liquidity. GMX utilizes a GLP mixed coin pool to meet users' demand for shorting or longing by creating an index, GLP, for the mixed coin pool, thus centralizing all provided liquidity in this single mixed coin pool. Unlike the order book and Uniswap's AMM model, its trading price relies on Oracles. Different from centralized exchanges' CFD trading, it has a higher degree of decentralization and holds 100% margin for CFD trades.

\*\*Background: Contract for Differences (CFD)\*\*

CFDs allow investment based on the price change of an asset between two points in time, with traders profiting from the rise or fall in the price of the underlying asset. Delivery is settled by the difference between the settlement price and the contract price, not by the underlying asset. CFD traders can control larger positions through leverage, allowing them to freely open positions for long or short in any market, typically on a margin, with costs including the spread paid during the trade and a holding cost for each trading day.

\*\*High-level Walkthrough\*\*

\*\*High-level walkthrough of deposit\*\*

Process for handling deposits:

a. Record token transfer: Record the transfer of tokens in the Vault.

b. Calculate execution costs: Determine the fees required to execute the deposit.

c. Store information: Includes the owner, callback, exchange path, unpacking, etc.

d. Execute the deposit (for custodians only):

- Verify custodian prices: Ensure the price used for the deposit is the accurate price sent by the custodian.

- Obtain market information: Determine the market intended for this deposit.

- Optional token exchange: If necessary, exchange the initial token for the final token to be deposited into the market.

- Price impact: Calculate the impact on the USD price.

- Incentive impact: Incentivize based on the price impact.

- Adjust pool amounts: Adjust the amounts in the pool based on the deposit operation.

\*\*High-level walkthrough of order\*\*

Process for handling orders:

a. Create an order:

- Record transfer: Record the transfer of tokens in the Vault.

- Obtain and store key information: Record the key information of the order.

b. Execute the order:

- Set order price: Assign a price to the order.

c. Process the order:

- Increase order: Increase the number or scale of the orders.

- Decrease order: Decrease the number or scale of the orders.

- Exchange orders: Use the deposit assets as liquidity for the exchange.

- Create position reference: Create a reference for the position.

- Fill positions: Fill in the position information at the corresponding location.

d. Handle collateral:

- Calculate the original price of the position (in USD): Determine the initial USD value of the position.

- Update the total open interest: Refresh the total open interest in the market.

- Verify open interest: Ensure there is enough support token available for payment.

- Compare the original price to the current price: Compare the original price of the position with the current price.

- Remove the order after updating: Remove the order after updating the information.

\*\*Main Research Questions\*\*

- \*\*How to Limit Global Position Size (Long and Short)\*\*

- In the `PositionManager`'s `\_validateMaxGlobalSize` method, there's a mapping for each token that limits the position size.

- `guaranteedUsd` represents the USD amount lent by LPs for leveraged positions, recording the difference between the token's position and the margin valued in USD.

- For long positions, the `guaranteedUsd` of that token + `sizeDelta` must be less than or equal to the maximum long position size.

- For short positions, the `globalShortSizes` of that token + `sizeDelta` must be less than the maximum short position size.

- `reservedAmounts[\_token]` <= `poolAmounts[\_token]`, where `reservedAmounts` records the total value of the position converted into the quantity of collateral token. That means GMX does not limit a single position but all global positions collectively. The total of all such collateral positions, `reservedAmounts`, cannot exceed the total quantity of that kind of token stored in the Vaults, `poolAmounts`.

- \*\*How to Manage Price (Price Feed, Fast Price Feed, Handling when Fast Price Feed Deviates from Chainlink Price by 0.5%)\*\*

- `PriceFeed` receives prices from the keeper, with different treatments for stablecoins.

- In `PriceFeed`, the contract takes three price samples. The keeper accepts the median price from exchanges, valid for five minutes. After the timeout, it uses Chainlink prices.

- The keeper is monitored by a watcher that continuously compares the median price with the keeper's. If the keeper's price does not match, the watcher forces a spread between the Chainlink price and the keeper's price.

- When the Chainlink price and the keeper's price are within a set percentage threshold, the keeper's price is used. Outside the threshold, a spread is triggered between the fast price and the Chainlink price.

- Compared to other price sources (e.g., Chainlink), the fast price can be updated more quickly by the updater in the contract through functions like `setPrices`, `setCompactedPrices`, or `setPricesWithBits`.

- In `FastPriceFeed`, the price from the mapping is typically returned without the spread, except under certain conditions when the spread is triggered:

- If the `FastPriceFeed` is inactive or times out, it returns a price with `spreadBasisPoints`.

- When the fast price deviates from the Chainlink price by more than a threshold, the larger or smaller price is returned based on `maximise`.

- When the watcher reports an issue and `favorFastPrice` is set, the same rules apply.

- \*\*How to Manage Positions (How to Liquidate, Price Calculation)\*\*

- \*\*Liquidation\*\*

- Liquidation operations are based solely on Chainlink prices in the `liquidatePosition` method.

- First, it checks for Private Mode (whitelist liquidation).

- Updates the cumulative funding rate for the staked token and trading pair token (`fundingRateFactor \* reservedAmounts[\_token]`) \* (intervals) / (poolAmount).

- Generates a position key from account, collateralToken, indexToken, and `\_isLong` and retrieves the corresponding position.

- Validates liquidation, returning different `liquidationState` for three scenarios: 1. No liquidation needed 2. Position loss greater than collateral, position fees greater than collateral, position fees plus liquidation fees greater than collateral all require liquidation 3. Over 100x leverage, but there is remaining collateral value after deducting losses.

- If it's the third scenario, execute `decreasePosition`, leaving a remaining value of collateral after position liquidation.

- Converts and adds liquidation fees to the `feeReserves` of the collateralToken.

- Updates the position and pool status, paying the fees.

- \*\*PnL Calculation\*\*

- For long positions, `nextAveragePrice = (nextPrice \* nextSize) / (nextSize + delta)`.

- For short positions, `nextAveragePrice = (nextPrice \* nextSize) / (nextSize - delta)`.

- PnL `delta` = (size \* priceDelta (spread between price and avgPrice) / avgPrice).

- \*\*How to Limit Opening Position Parameters (What Situations Need Verification, and What Scenarios Do They Correspond To, Such as LP Side or User Side)\*\*

- \*\*LP Side\*\*

- For going long, collateral must be non-stablecoins such as WBTC, WETH, UNI, etc., and the trading pair must match the collateral.

- For going short, collateral must be stablecoins like USDC, USDT, DAI, etc., with trading pairs limited to WETH-For Short Positions:

- Ensure collateral must be a stablecoin.

- `\_validate(stableTokens[\_collateralToken]);`

- Ensure the index token is not a stablecoin to prevent arbitrage between stablecoins.

- `\_validate(!stableTokens[\_indexToken]);`

- Check if the index token can be shorted to prevent manipulation.

- `\_validate(shortableTokens[\_indexToken]);`

\*\*GLP Design and Potential for Arbitrage (Impact of AUM on LP Price)\*\*

GLP provides liquidity for leverage trading by staking any token in the liquidity pool, earning 70% of the funding rate income from trading and funding fees. Its price is determined by the value of all tokens in the pool and the profit or loss of all open positions (AUM / supply).

\*\*AUM Calculation\*\*

- For stablecoins, sum the USD value: poolAmount \* price / 10 \*\* decimals.

- For non-stablecoins, calculate short position profit/loss, guaranteedUsd, and calculate the USD value of the non-reserved part (poolAmount - reservedAmount) of the pool, incorporating these into AUM.

\*\*Impact of AUM on GLP Price:\*\*

- When traders lose, AUM increases. The value of each unit of asset in the pool rises. Consequently, the price of GLP tokens also increases as each GLP token represents more asset value.

- Conversely, when AUM decreases (e.g., traders profit), the value of each unit of asset in the pool decreases. Thus, the price of GLP tokens also decreases.

\*\*Opening, Closing Positions, and Swap Process\*\*

- \*\*Global Exec Fee\*\*

- `PositionRouter -> WETH -> PositionRouter -> keeper`

- \*\*Engaging Leverage\*\*

- Solidity Code:

```solidity

// Router.sol

function approvePlugin(address \_plugin) external {

approvedPlugins[msg.sender][\_plugin] = true;

}

```

- \*\*FastPriceFeed\*\*

- Solidity Code:

```solidity

function setPricesWithBitsAndExecute(

address \_positionRouter,

uint256 \_priceBits, // Price information

uint256 \_timestamp,

uint256 \_endIndexForIncreasePositions, // End index for position requests

uint256 \_endIndexForDecreasePositions,

uint256 \_maxIncreasePositions, // Maximum number of positions that can be adjusted

uint256 \_maxDecreasePositions

) external onlyUpdater {

// Update prices and timestamp

\_setPricesWithBits(\_priceBits, \_timestamp);

// Execute position changes

positionRouter.executeIncreasePositions(\_endIndexForIncreasePositions, payable(msg.sender));

positionRouter.executeDecreasePositions(\_endIndexForDecreasePositions, payable(msg.sender));

}

```

- \*\*Increasing Long Position (ETH -> long BTC)\*\*

- This is a coin-denominated operation, where profits or losses are calculated in BTC, affected by BTC prices.

- Request Increase Long: `user(ETH) -> WETH -> PositionRouter -> createIncreasePositionETH`

- Logic: Transfer to the PositionRouter contract, record the request.

- Increase Long (keeper call): `PositionRouter(WETH) -> Vault(BTC) -> PositionRouter(BTC) -> Vault`

- \*\*Decreasing Long Position\*\*

- Request Decrease Long: `PositionRouter -> createDecreasePosition`

- Logic: Transfer fees into the PositionRouter contract, verify path, record the request.

- Decrease Long (keeper call): `Vault(BTC) -> PositionRouter(BTC) -> user`

- \*\*Increasing Short Position (BTC -> short BTC)\*\*

- This is a USD-denominated operation, essentially selling BTC, providing an equivalent USDC as collateral. If BTC's price falls, one can buy back the same amount of BTC at a lower price and repay the loan, earning the difference.

- Request Increase Short: `User(WBTC) -> PositionRouter -> createIncreasePosition`

- Logic: Transfer fees and ERC20 into the PositionRouter contract, record the request.

- Increase Short (keeper call): `PositionRouter(BTC) -> Vault(USDC) -> PositionRouter(USDC) -> Vault`

- \*\*Decreasing Short Position\*\*

- Request Decrease Short: `PositionRouter -> createDecreasePosition`

- Decrease Short (keeper call): `Vault(USDC) -> PositionRouter(USDC) -> user`

- \*\*Swap\*\*

- `Router -> swapTokensToETH`

- `user(BTC) -> Vault(WETH) -> Router -> WETH(ETH) -> Router(ETH) -> user`

\*\*Fees and Calculations\*\*

- In contracts, prices are represented in USD to 30 decimal places.

- \*\*Calculating Position Fees\*\*

- `intervals = (timestamp - lastFundingTimes[token]) / 1h`

- `next funding rate = factor \* reservedAmount \*

\*\*Architecture\*\*

- \*\*Vault\*\*

- Handles deposits and trading.

- \*\*Routers\*\*

- \*\*OrderBook\*\* and \*\*PositionManager\*\* manage token transfers, wrapping/unwrapping for the Vault.

- \*\*Price Feeds\*\*

- Process submissions from price feed keepers.

- \*\*GMX\*\*

- Governance ERC20 token, rewards for staking.

- \*\*GLP\*\*

- LP token minted in the liquidity pool.

\*\*Modules\*\*

- \*\*Vault (Mixed Coin Pool)\*\*

- As a mixed coin pool, it assigns target ratios to each coin to define an index: GLP. The accounting in the mixed coin pool is represented using USDG ($1 value).

- Solidity Code:

```solidity

// tokenWeights allows customization of index composition

mapping (address => uint256) public override tokenWeights;

```

- Target Ratios and Current Weights:

- WETH: 25% -- 28.65%

- WBTC: 15% -- 14.26%

- LINK: 5% -- 2.49%

- UNI: 1% -- 0.81%

- USDC: 30% -- 35.15%

- USDT: 9% -- 6.06%

- DAI: 12% -- 10.98%

- MIM: 1% -- 0%

- FRAX: 2% -- 1.58%

- Retail traders can open long or short positions, providing non-stablecoins for long positions or stablecoins for short positions to the Vault, with profit/loss settlement and collateral in the same currency type.

- Any user can swap by providing tokenA to the vault and exchanging it for tokenB based on the oracle's price. It's akin to the user selling tokenA to the system in exchange for the system's internally accounted usdg, then using that usdg to buy tokenB. At this point, tokenA flows in, and tokenB flows out.

\*\*Building Positions\*\*

- \*\*For Retail Investors Opening Long Positions\*\*:

- When a retail investor opens a 10x leveraged long position, essentially, the investor only contributes 1 ETH, while the LPs in the vaults lend 9 ETH to the investor.

- Effectively, the investor exchanges 1 ether for an equivalent 3,000 USDG in the vaults.

- Then, LPs in the vaults lend an additional 27,000 USDG to the investor, making up a total position of 30,000 USDG.

- At this point, the average price of this position is 3,000 USDG.

- The opening fee (0.1%), including funding costs (0 USD), amounts to 30 USDG, equivalent to 0.01 ETH.

- This necessitates recording the debt of 27,030 USDG (guaranteedUsd) lent by LPs in the vaults to the investor for WETH.

- Also, the actual transferred WETH amount by the user to the vaults, i.e., poolAmounts (WETH ⇒ 0.99 ETH), needs to be recorded.

- It's imperative that the reserve WETH amount required to liquidate all retail positions must be less than the total WETH amount held in the vaults.

- \*\*For Retail Investors Opening Short Positions\*\*:

- When a retail investor uses 3,000 USDC to open a 10x leveraged short position, it means the LPs in the vaults lend the investor 27,000 USDC.

- In essence, the investor exchanges 3,000 USDC for an equivalent 3,000 USDG in the vaults.

- Then, LPs in the vaults lend an additional 27,000 USDG to the investor, constituting a total position of 30,000 USDG.

- This position's average price is set at 3,000 USDG. At this point, the opening fee, including funding costs (0 USD), is 30 USDG.

- It's necessary to ensure that there are sufficient USDC reserves in the vaults to liquidate all retail positions, i.e., reserveAmounts (USDC ⇒ 30,000 ETH) must be less than the total USDC amount held in the vaults.

\*\*Swap\*\*

- The swap module allows for exchanging tokens within the Vault.

- Solidity Code:

```solidity

function swap(

address \_tokenIn, // Incoming token

address \_tokenOut, // Outgoing token

address \_receiver // Receiver

) external override nonReentrant returns (uint256) {

// Verification... ensure the token is whitelisted

// Calculate rates

updateCumulativeFundingRate(\_tokenIn, \_tokenIn);

updateCumulativeFundingRate(\_tokenOut, \_tokenOut);

// Incoming contract amount

uint256 amountIn = \_transferIn(\_tokenIn);

// Calculate exchange prices

uint256 priceIn = getMinPrice(\_tokenIn);

uint256 priceOut = getMaxPrice(\_tokenOut);

// Outgoing contract amount

uint256 amountOut = amountIn.mul(priceIn).div(priceOut);

amountOut = adjustForDecimals(amountOut, \_tokenIn, \_tokenOut);

// Adjust usdg amounts between swaps

uint256 usdgAmount = amountIn.mul(priceIn).div(PRICE\_PRECISION);

usdgAmount = adjustForDecimals(usdgAmount, \_tokenIn, usdg);

// Calculate base fee rate

uint256 feeBasisPoints = vaultUtils.getSwapFeeBasisPoints(\_tokenIn, \_tokenOut, usdgAmount);

uint256 amountOutAfterFees = \_collectSwapFees(\_tokenOut, amountOut, feeBasisPoints);

// Adjust usdg and token amounts in the pool

\_increaseUsdgAmount(\_tokenIn, usdgAmount);

\_decreaseUsdgAmount(\_tokenOut, usdgAmount);

\_increasePoolAmount(\_tokenIn, amountIn);

\_decreasePoolAmount(\_tokenOut, amountOut);

\_validateBufferAmount(\_tokenOut);

\_transferOut(\_tokenOut, amountOutAfterFees, \_receiver);

// Emit event...

return amountOutAfterFees;

}

```

\*\*Liquidations\*\*

- Liquidations involve calling the `validateLiquidation` method to return a `liquidationState`. If the value is 2 (exceeding leverage), a position decrease is executed. Otherwise, a liquidation is performed, requiring updates to GuaranteedUsd and PoolAmount for long positions.

\*\*Managing Positions\*\*

The management of positions within the GMX platform involves several key processes, such as opening positions, maintaining margin requirements, adjusting positions (both increasing and decreasing), and ultimately, liquidation in case the position becomes undercollateralized.

\*\*Opening Positions\*\*

- When a user decides to open a position, whether long or short, they must provide collateral in the form of a specified token. This collateral is then used to borrow additional funds from the liquidity pool to enter a leveraged position.

- For long positions, users typically post collateral in non-stablecoins (like ETH) and aim to profit from an increase in the price of the underlying asset.

- For short positions, the collateral is usually in stablecoins, with the user betting on a decline in the price of the underlying asset.

\*\*Margin Requirements\*\*

- The platform requires users to maintain a certain margin level to keep their positions open. This is to ensure that there is enough collateral to cover potential losses.

- Margin fees, including funding costs and potential swap fees, are calculated at the time of opening the position and periodically throughout its duration.

- Positions are monitored continuously to ensure they meet the margin requirements. If a position's value falls below the required margin level due to market movements, the user may be called upon to add more collateral.

\*\*Adjusting Positions\*\*

- Users can adjust their positions by either increasing (adding more to the position) or decreasing (closing out a portion of the position) their exposure.

- Increasing a position involves adding more collateral and potentially taking on more debt from the liquidity pool, thus increasing the size of the position.

- Decreasing a position involves selling off a portion of the underlying asset or paying back a portion of the borrowed funds, thus reducing the size of the position.

- These adjustments can be made to either take further advantage of favorable market movements or to reduce exposure in response to market changes.

\*\*Liquidation\*\*

- If a position becomes too risky, meaning it no longer meets the margin requirements and the user fails to add more collateral, it may be subject to liquidation.

- Liquidation is the process where the platform forcibly closes the position to ensure that the debt owed to the liquidity pool is repaid.

- This process is triggered automatically by the smart contract when certain conditions are met, such as the collateral value falling below a specified threshold.

- The liquidation process involves selling the collateral at the current market price. If the proceeds from the sale are insufficient to cover the debt, the shortfall is covered by the platform's insurance fund or through other mechanisms designed to protect the liquidity pool.

\*\*Smart Contract Implementation\*\*

- The entire process of managing positions is governed by smart contracts, which execute automatically based on the code's logic and the prevailing market conditions.

- These contracts handle the transfer of tokens, calculation of fees, adjustment of position sizes, and the execution of liquidations, among other functions.

- The smart contract code ensures that all actions are performed transparently and securely, without the need for manual intervention by the platform's operators.

\*\*Example Solidity Implementation for Managing Positions\*\*

```solidity

// Simplified example of increasing a position

function increasePosition(

address \_account,

address \_collateralToken,

uint256 \_amount,

bool \_isLong

) external {

// Verify the account, collateral, and amount

// Calculate the required fees and new position size

// Adjust the position's collateral and size in the smart contract's state

// Emit an event for the position increase

}

// Simplified example of liquidating a position

function liquidatePosition(

address \_account,

bool \_isLong

) external {

// Determine if the position meets the criteria for liquidation

// Calculate the amount to be liquidated and the fees

// Close the position and update the smart contract's state

// Transfer the collateral to cover the debt and fees

// Emit an event for the liquidation

}

```

These simplified examples illustrate the core logic behind position management and liquidation processes within a leveraged trading platform like GMX. The actual implementation may include additional checks, balances, and features to ensure platform stability and user security.

The process of managing and updating the average price of global positions within a platform like GMX involves intricate mechanisms, particularly when it comes to leveraged positions like longs and shorts. This is crucial for maintaining fair and accurate accounting of positions' values, ensuring the system remains balanced and equitable for all users. Here's how the process typically works, focusing on the calculation of average prices for positions:

### For Long Positions:

When a user opens or increases a long position, the average price of that position is recalculated to reflect the additional amount at the current market price. The process is as follows:

1. \*\*Initial Long Position\*\*: If this is the user's first long position in a specific token, the initial average price of the position is set to the `maxPrice` of the `indexToken`. This `maxPrice` represents the highest price of the token according to the platform's price feed at the moment of the transaction.

```solidity

uint256 price = getMaxPrice(\_indexToken);

```

2. \*\*Subsequent Increases in Long Position\*\*: If the user is adding to an existing long position, the new average price needs to be calculated to account for the additional purchase at the current price. This is done using the formula:

```plaintext

New Average Price = (Current Average Price \* Current Position Size + Max Price \* Size Delta) / (Current Position Size + Size Delta)

```

Where:

- `Current Average Price` is the position's average price before the increase.

- `Current Position Size` is the total size of the position before the increase.

- `Max Price` is the maximum price of the token at the time of the increase.

- `Size Delta` is the amount by which the position is being increased.

### For Short Positions:

For short positions, the process is analogous, but the pricing is based on the `minPrice` of the `indexToken`, reflecting the lowest price of the token according to the platform's price feed at the moment of the transaction.

1. \*\*Initial Short Position\*\*: The initial average price of a new short position is set to the `minPrice` of the `indexToken`.

2. \*\*Subsequent Increases in Short Position\*\*: When increasing a short position, the new average price is calculated using a similar formula but with the `minPrice`:

```plaintext

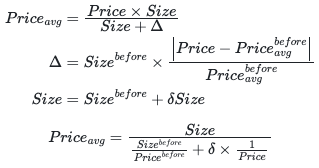
New Average Price = (Current Average Price \* Current Position Size + Min Price \* Size Delta) / (Current Position Size + Size Delta)

```

### Global Position Average Price Adjustment:

The platform also needs to adjust the global average price of all positions (both long and short) based on the total global position size. This ensures that the platform's liquidity and risk are managed effectively, and the pricing reflects the aggregate of individual positions' values. This adjustment is crucial for risk management, especially in volatile markets where price discrepancies can lead to significant imbalances.

The implementation in Solidity might involve iterating over all positions or maintaining a running average that gets updated with each position increase or decrease. The exact method can vary based on the platform's architecture and specific requirements for accuracy, gas efficiency, and real-time updates.



The Solidity function `getNextAveragePrice` you've shared is designed to calculate the next average price of a position after a change in its size, whether through increasing or decreasing the position. This calculation is crucial for accurately tracking the performance and value of a position over time, especially in a trading platform that supports leveraged positions like longs and shorts.

Here's a detailed explanation of how this function works and its role in managing positions and funds:

### Function Overview

- \*\*Parameters\*\*:

- `\_indexToken`: The token in which the position is held.

- `\_size`: The current size of the position before the change.

- `\_averagePrice`: The current average price of the position.

- `\_isLong`: A boolean indicating whether the position is long (true) or short (false).

- `\_nextPrice`: The price at which the position size is changing.

- `\_sizeDelta`: The amount by which the position size is changing.

- `\_lastIncreasedTime`: The timestamp of the last time the position size was increased.

### Calculating the Next Average Price

1. \*\*Determining Profit or Loss\*\*: First, the function calculates whether the position is currently in profit or loss by comparing the market price (`\_nextPrice`) to the position's average price (`\_averagePrice`). This is achieved through the `getDelta` function, which returns a boolean indicating whether the position has a profit and the delta (the difference in value due to the price change).

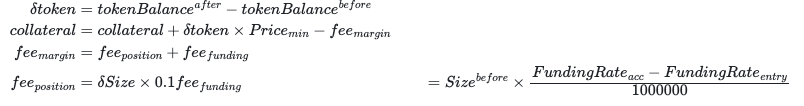
2. \*\*Adjusting for Position Size Change\*\*: The function then calculates the new size of the position (`nextSize`) by adding the size change (`\_sizeDelta`) to the current size (`\_size`).

3. \*\*Calculating the Divisor\*\*: The divisor for calculating the new average price depends on whether the position is in profit and whether it is a long or short position. The divisor adjusts the position size based on the profit or loss to ensure that the new average price reflects the value of the position accurately.

4. \*\*Calculating the New Average Price\*\*: Finally, the new average price is calculated by multiplying the next price (`\_nextPrice`) by the new position size (`nextSize`) and dividing by the divisor. This calculation adjusts the average price to reflect the impact of the position size change at the new market price.

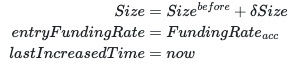
### Updating Position & Funds Fee

After calculating the new average price, the trading platform must update the position's collateral value in USD. This involves adjusting the collateral value to account for the change in position size and the associated fees, including funding fees and any other costs related to maintaining the position.



### Updating Reserve

The platform also needs to update the reserve amounts to ensure that there are sufficient funds to cover the positions. This involves adjusting the reserved amounts of the collateral and index tokens to reflect the new size and value of the positions, ensuring that the platform can meet its obligations to traders.



### Implementation in Solidity

The `getNextAveragePrice` function exemplifies the complex calculations required to accurately track and manage leveraged positions in a decentralized finance (DeFi) trading platform. By carefully updating the average price and adjusting reserves according to position changes, the platform can maintain a balanced and secure trading environment for its users.

Updating a position's entry, size, and time within a trading platform that supports leveraged trades involves a set of carefully calculated steps. These updates are crucial for tracking the performance of each position accurately and ensuring that the platform can manage risk effectively. Here's an overview of how these updates might be implemented in Solidity:

### Updating Position Entry

The position entry refers to the initial price at which the position was opened or last adjusted. This value is essential for calculating profits or losses and determining the position's performance over time.

1. \*\*Calculate the New Average Price\*\*: As discussed, the new average price after a position size change is calculated based on whether the position is a long or a short, the market price at the time of the change, and the direction of the profit or loss. This new average price becomes the new entry price for the position.

### Updating Position Size

The size of the position indicates the amount of the underlying asset or derivative being traded. Position size changes when a trader either adds to the position (increasing size) or closes part of the position (decreasing size).

1. \*\*Increase Position Size\*\*: When adding to a position, the new size is calculated by adding the delta (the amount being added) to the existing position size.

2. \*\*Decrease Position Size\*\*: When closing part of a position, the new size is calculated by subtracting the delta (the amount being closed) from the existing position size.

### Updating Timestamp

Updating the timestamp whenever a position is adjusted is crucial for several reasons, such as calculating funding rates, tracking position age, and enforcing certain trading rules.

1. \*\*Record Last Increased Time\*\*: Whenever a position size is increased, the current block timestamp (`block.timestamp` in Solidity) is recorded as the `lastIncreasedTime`. This timestamp is used for calculations that depend on the time elapsed since the last increase.

### Solidity Implementation Example

Here's a simplified example of how these updates might be coded in Solidity:

```solidity

function adjustPosition(

address \_account,

address \_indexToken,

bool \_isLong,

uint256 \_sizeDelta,

bool \_increaseSize

) external {

// Unique key to identify the position

bytes32 positionKey = keccak256(abi.encodePacked(\_account, \_indexToken, \_isLong));

Position storage position = positions[positionKey];

uint256 nextPrice = \_isLong ? getMaxPrice(\_indexToken) : getMinPrice(\_indexToken);

uint256 nextSize = \_increaseSize ? position.size.add(\_sizeDelta) : position.size.sub(\_sizeDelta);

// Update the average price (entry) of the position

uint256 nextAveragePrice = getNextAveragePrice(

\_indexToken,

position.size,

position.averagePrice,

\_isLong,

nextPrice,

\_sizeDelta,

position.lastIncreasedTime

);

// Update position details

position.averagePrice = nextAveragePrice;

position.size = nextSize;

if (\_increaseSize) {

position.lastIncreasedTime = block.timestamp;

}

// Additional logic to handle collateral, fees, and validations

// ...

}

```

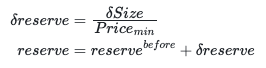
In this example, the `adjustPosition` function updates the position's entry (average price), size, and last increased time based on the inputs provided. Note that this example is simplified; a real-world implementation would include additional logic for handling collateral, fees, validations, and events.

These updates ensure that each position reflects the latest market conditions and trader actions, which is essential for maintaining a fair and transparent trading platform.

Updating the `reserveAmount` of a position and maintaining the ledger for `guaranteedUsd` and `poolAmount` are critical aspects of managing leveraged trading positions within a decentralized finance (DeFi) platform. Here's how these updates generally work:

### Updating Reserve Amount

The `reserveAmount` represents the total value converted into the collateral token that is set aside to ensure the platform can cover the position. This amount needs to be accurately tracked and updated whenever a position is opened, increased, or decreased.



1. \*\*Increase in Position Size\*\*: When a position size increases, the `reserveAmount` should also increase proportionally to ensure adequate collateral is available to cover the new size of the position.

2. \*\*Decrease in Position Size\*\*: Conversely, when a position size decreases, the `reserveAmount` should decrease accordingly, reflecting the reduced need for collateral.

### Updating the Ledger

#### GuaranteedUsd Ledger

The `guaranteedUsd` ledger tracks the difference between the position's value in the token and the collateral valued in USD. This is essential for managing the platform's risk and ensuring liquidity providers (LPs) have sufficient coverage.

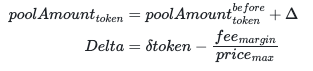


1. \*\*Opening or Increasing a Position\*\*: When a new position is opened or an existing position is increased, the `guaranteedUsd` for the corresponding token needs to be updated to reflect the added value of the position in USD.

2. \*\*Closing or Decreasing a Position\*\*: When a position is closed or decreased, the `guaranteedUsd` should be adjusted downwards to reflect the reduction in exposure.

#### PoolAmount Ledger

The `poolAmount` ledger tracks the quantity of each collateral token held in the pool. It is crucial for the platform to manage its liquidity and collateral levels.



1. \*\*Excluding Fees from PoolAmount\*\*: Fees, including margin fees, are not considered part of the `poolAmount` since they are deducted from the position's collateral value. Therefore, when fees are charged, the corresponding token quantity needs to be subtracted from the `poolAmount`.

2. \*\*Adjustments for Transactions\*\*: Whenever a transaction occurs that affects the position's collateral (e.g., opening, increasing, decreasing a position, or charging fees), the `poolAmount` should be adjusted to accurately reflect the current state of the pool's liquidity.

### Solidity Implementation Example

Here's a simplified example of how these updates might be implemented in Solidity:

```solidity

function updateReserveAndLedgers(

address \_token,

uint256 \_sizeDeltaUsd,

uint256 \_feeUsd,

bool \_isIncrease

) external {

uint256 feeTokens = usdToToken(\_token, \_feeUsd);

if (\_isIncrease) {

// Increase position size

guaranteedUsd[\_token] = guaranteedUsd[\_token].add(\_sizeDeltaUsd);

poolAmounts[\_token] = poolAmounts[\_token].add(usdToToken(\_token, \_sizeDeltaUsd)).sub(feeTokens);

} else {

// Decrease position size

guaranteedUsd[\_token] = guaranteedUsd[\_token].sub(\_sizeDeltaUsd);

poolAmounts[\_token] = poolAmounts[\_token].sub(usdToToken(\_token, \_sizeDeltaUsd)).add(feeTokens);

}

// Adjust reserveAmount accordingly

// ...

}

```

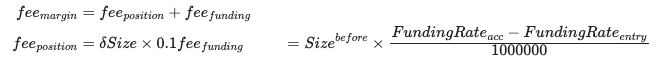
In this example, `updateReserveAndLedgers` function handles the adjustments to `guaranteedUsd` and `poolAmounts` ledgers based on the position size change and fees charged. Note that the actual implementation would involve more complex logic, including validations, event emissions, and handling edge cases.

Accurate ledger management is essential for maintaining platform integrity, ensuring liquidity, and managing risk effectively.

The provided Solidity functions detail the mechanisms for collecting margin fees, validating liquidation, and calculating rates within a decentralized finance (DeFi) platform. Let's break down each function to understand their purpose and functionality:

### Collecting Margin Fees

The `\_collectMarginFees` function calculates and collects the fees associated with adjusting a position's size, incorporating both the position fee and the funding fee.



1. \*\*Calculate Position Fee\*\*: This fee is determined by the size of the position adjustment (`\_sizeDelta`). It's typically a small percentage of the position size, adjusted for leverage.

2. \*\*Calculate Funding Fee\*\*: The funding fee compensates liquidity providers for the cost of funding the leverage provided to traders. It's calculated based on the size of the position (`\_size`), the difference between the current funding rate and the rate at the time the position was entered (`\_entryFundingRate`), and is prorated to the size of the position adjustment.

3. \*\*Fee Payment\*\*: The total fee in USD (`feeUsd`) is converted into the collateral token amount (`feeTokens`), which is then added to the `feeReserves` for the collateral token. This mechanism ensures that fees are collected in the native token of the position's collateral, maintaining the liquidity and integrity of the platform's financial ecosystem.

### Validating Liquidation

The `validateLiquidation` function checks if a position meets the conditions for liquidation based on its current state and market conditions.

1. \*\*Profit/Loss and Delta\*\*: It first determines if the position is in profit and calculates the `delta`, representing the potential loss relative to the position's current collateral value.

2. \*\*Margin Fees\*\*: It calculates the total margin fees associated with the position, including both the funding fee and the position fee.

3. \*\*Collateral and Liquidation Conditions\*\*: The function then checks various conditions to determine if liquidation is warranted:

- If losses exceed the collateral value.

- If the remaining collateral after losses is less than the total margin fees.

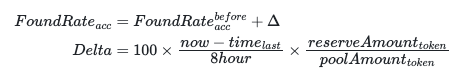
- If adding the platform's liquidation fee to the margin fees exceeds the remaining collateral.

- If the position's leverage exceeds the platform's maximum allowed leverage after accounting for losses and fees.

If any of these conditions are met, the function signals that liquidation may proceed, returning the appropriate flags and fee amounts to guide the liquidation process.

### Rate Calculation and Cumulative Funding Rate

The cumulative funding rate is an essential aspect of managing leveraged positions, impacting the calculation of funding fees over time. This rate evolves based on the supply and demand dynamics of leverage within the platform, reflecting the cost of borrowing or lending assets to open or maintain leveraged positions.



1. \*\*Accumulation Over Time\*\*: The cumulative funding rate adjusts periodically based on the platform's funding interval. It accounts for the continuous time-based accrual of funding costs, ensuring that liquidity providers are compensated for the risk and capital they provide to leveraged traders.

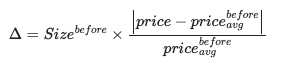
2. \*\*Impact on Positions\*\*: For traders, this rate influences the long-term cost of maintaining open positions. Positions held for longer periods may incur higher funding fees, especially in volatile markets or when the demand for leverage is high.

These functions underscore the complexity of managing financial interactions on a DeFi trading platform, balancing the need to incentivize liquidity provision while ensuring that traders can execute leveraged strategies efficiently and fairly.

Closing PnL:

For users, they have the option to either partially close their positions or close them entirely. When partially closing a position, users can specify the amount of collateral they wish to withdraw, which is valued in USD.

For partially closing a long position: The unrealized profit or loss (delta) is calculated using the index token corresponding to the position.

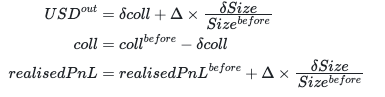


Updating the `realisedPnL` (realized Profit and Loss) and the collateral (`coll`) values, where `USDout` is the USD value transferred from the Vault to the user upon closing a position, involves a couple of key steps. Here's a simplified explanation of the process, focusing on a scenario where the user makes a profit (i.e., the exit price is greater than or equal to the average entry price of the position):

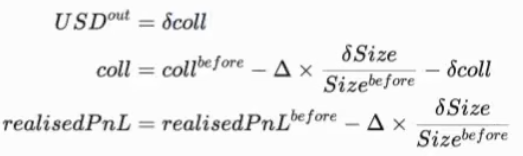
1. \*\*Calculate Realized PnL\*\*:

- Realized PnL is the actual profit or loss made on the position at the time of closing, compared to the entry price. It is calculated based on the difference between the market price at the time of closing and the average entry price, multiplied by the size of the position being closed.

- If the user is in profit (`price >= price (before avg)`), the realized PnL is positive. This indicates that the exit price is higher than the average entry price of the position.



If the user is in not profit ：price < price (before avg)



2. \*\*Update Collateral (`coll`) Value\*\*:

- The collateral value needs to be updated to reflect the change in the user's balance after the position is closed. This includes adjusting for the realized PnL.

- If the user profits from the trade, the collateral (`coll`) value increases by the amount of the profit. Conversely, if the user incurs a loss, the collateral value decreases by the amount of the loss.



3. \*\*Consider `USDout` in the Calculation\*\*:

- `USDout` represents the USD value that the Vault transfers to the user when closing the position. This amount should reflect the value of the position being closed at the current market price, including any realized profits or covering any realized losses.

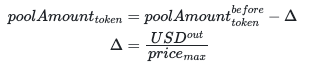


- The calculation of `USDout` must take into account the size of the position being closed and the current market price of the underlying asset or index token. It should also adjust for any fees or charges applicable upon closing.



### Example in Solidity

Here's a simplified example to illustrate how the realized PnL and collateral values might be updated in Solidity:



```solidity

function updateRealisedPnLAndCollateral(

address account,

uint256 positionSize,

uint256 avgEntryPrice,

uint256 exitPrice,

bool isProfit

) external {

// Simplified calculation; in practice, consider fees and exact price calculations

uint256 realisedPnL;

if (isProfit) {

realisedPnL = (exitPrice - avgEntryPrice) \* positionSize;

// Update user's collateral value by adding the profit

collaterals[account] = collaterals[account].add(realisedPnL);

} else {

realisedPnL = (avgEntryPrice - exitPrice) \* positionSize;

// Update user's collateral value by subtracting the loss

collaterals[account] = collaterals[account].sub(realisedPnL);

}

// Assume USDout calculation is done here based on exitPrice and positionSize

uint256 USDout = calculateUSDout(exitPrice, positionSize);

// Transfer USDout to the user from the Vault

vault.transferUSDoutToUser(account, USDout);

// Emit an event or log for the update

emit RealisedPnLAndCollateralUpdated(account, realisedPnL, collaterals[account], USDout);

}

```

This example highlights the key steps in updating the realized PnL and the collateral values when a user closes a position, whether fully or partially. The actual implementation would need to account for the complexity of the market, fees, and the specific mechanisms of the DeFi platform.

The process of closing positions on a decentralized finance (DeFi) platform involves careful consideration of fees, adjustments to the position's collateral, and updates to various platform metrics like `guaranteedUSD` and `PoolAmount`. Here’s a breakdown of these operations, including simplified Solidity code examples to illustrate these actions:

### Deducting Fees

When a user closes a position and receives `USDout`, the process for handling fees depends on the amount of `USDout` relative to the applicable fees:

- \*\*If `USDout` is greater than the fees\*\*, the fees are deducted directly from `USDout`.

- \*\*If `USDout` is smaller than the fees\*\*, the difference is deducted from the user's collateral.

This mechanism ensures that the platform always collects the necessary fees while adjusting the user's return or collateral accordingly.

### Updating the Position and Entry Funding Rate

After closing or adjusting a position, it's crucial to update the position's status, including its size and the entry funding rate. For long positions, this also includes updating the `guaranteedUSD` mapping to reflect changes in the guaranteed dollar value of the position.

### Before Transferring `USDout` to the User

Before finalizing the transaction and transferring `USDout` to the user, the platform must calculate the delta of the pool tokens resulting from the position's closure and update the `PoolAmount` ledger accordingly. This step ensures that the liquidity pool's records accurately reflect the current state of all assets within it.

### Solidity Implementation Examples

#### Withdrawing Fees

The `withdrawFees` function allows the platform to transfer collected fees from a specified token to a designated receiver, usually as part of platform revenue or for redistribution among liquidity providers.

```solidity

function withdrawFees(

address \_token,

address \_receiver

) external override returns (uint256) {

\_onlyGov(); // Ensure only governance or authorized entities can call

uint256 amount = feeReserves[\_token];

if(amount == 0) { return 0; }

feeReserves[\_token] = 0;

\_transferOut(\_token, amount, \_receiver);

return amount;

}

```

#### Decreasing Position Collateral

The `\_decreasePosition` function manages the process of reducing a position's size, adjusting collateral, and handling the transfer of `USDout` to the user. This includes deducting fees and updating position metrics.

```solidity

function \_decreasePosition(

address \_account,

address \_collateralToken,

address \_indexToken,

uint256 \_collateralDelta,

uint256 \_sizeDelta,

bool \_isLong,

address \_receiver

) private returns (uint256) {

// Assume position retrieval and validation logic here

// Check the position size against \_sizeDelta

if (position.size != \_sizeDelta) {

// Additional logic for partial position decrease

if (\_isLong) {

// Update position metrics for a long position

}

// More adjustments as needed

} else {

// Logic to remove the position if fully closed

}

// Further logic to handle transfers and ledger updates

}

```

These functions highlight key operations involved in managing and closing positions within a DeFi trading platform. They underscore the importance of accurate accounting for fees, collateral, and liquidity pool metrics to maintain the platform's integrity and operational efficiency.

The Router contracts in a decentralized finance (DeFi) platform like GMX serve as interfaces between users (traders) and the platform's Vault, facilitating various operations such as token transfers to the Vault, wrapping of native tokens, and more. Let's delve into the details of these operations, focusing on the PositionRouter which handles market orders for increasing and decreasing positions, both long and short, and how it interacts with keepers for price execution within a specified slippage range.

### Handling Increase Position Requests

When a user wants to increase their position, they initiate a request that involves transferring the necessary tokens to the Vault, specifying the desired parameters for the trade, including the amount to be invested (`\_amountIn`), the minimum acceptable outcome of the trade (`\_minOut`), the size of the position increase (`\_sizeDelta`), and whether the position is long (`\_isLong`).

The request is then processed by the keeper, which fetches the current index price from an aggregator. Based on this price, the keeper attempts to execute the position. If the trade cannot be executed within the acceptable slippage (`\_acceptablePrice`), it's rolled back to prevent undesired losses for the user.

#### Solidity Implementation for Increase Position

```solidity

function createIncreasePosition(

address[] memory \_path, // Trading pair

address \_indexToken,

uint256 \_amountIn,

uint256 \_minOut,

uint256 \_sizeDelta,

bool \_isLong,

uint256 \_acceptablePrice,

uint256 \_executionFee,

bytes32 \_referralCode,

address \_callbackTarget

) external payable nonReentrant returns (bytes32) {

// Execution fee check

require(\_executionFee >= minExecutionFee, "fee");

require(msg.value == \_executionFee, "val");

// Asset conversion check

require(\_path.length == 1 || \_path.length == 2, "len");

\_transferInETH(); // Transfers ETH in case of native token

\_setTraderReferralCode(\_referralCode); // Sets referral code

// Handles token transfer for increasing position

if (\_amountIn > 0) {

IRouter(router).pluginTransfer(\_path[0], msg.sender, address(this), \_amountIn);

}

// Update state...

}

```

This function is responsible for handling the preliminary operations required to increase a position, such as fee verification and token transfer.

### Handling Decrease Position Requests

Similarly, when a user wants to decrease their position, they can specify the amount of collateral they wish to withdraw, valued in USD, and initiate a request for the position decrease.

#### Solidity Implementation for Decrease Position

```solidity

function executeDecreasePosition(

bytes32 \_key,

address payable \_executionFeeReceiver

) public nonReentrant returns (bool) {

// Read and verify Request

DecreasePositionRequest memory request = decreasePositionRequests[\_key];

if (request.account == address(0)) { return true; }

// Validation based on position delay time and keeper request

bool shouldExecute = \_validateExecution(request.blockNumber, request.blockTime, request.account);

if (!shouldExecute) { return false; }

// Update state...

delete decreasePositionRequests[\_key];

// Verify position price, use TimeLock to update global shorts, execute decrease

uint256 amountOut = \_decreasePosition(request.account, request.path[0], request.indexToken, request.collateralDelta, request.sizeDelta, request.isLong, address(this), request.acceptablePrice);

// Handle trading pair...

if (amountOut > 0) {

if (request.path.length > 1) {

IERC20(request.path[0]).safeTransfer(vault, amountOut);

amountOut = \_swap(request.path, request.minOut, address(this));

}

if (request.withdrawETH) {

\_transferOutETHWithGasLimitFallbackToWeth(amountOut, payable(request.receiver));

} else {

IERC20(request.path[request.path.length - 1]).safeTransfer(request.receiver, amountOut);

}

}

\_transferOutETHWithGasLimitFallbackToWeth(request.executionFee, \_executionFeeReceiver);

// Event...

\_callRequestCallback(request.callbackTarget, \_key, true, false);

return true;

}

```

This function manages the process for decreasing a position, including transferring out the adjusted collateral to the user or their specified receiver, handling fee deductions, and ensuring the position is updated accordingly in the platform's records.

Both increasing and decreasing positions involve complex interactions between the user's requests, the platform's liquidity management (handled by the Vault), and market conditions (managed through price feeds and keepers). These operations are crucial for maintaining the platform's liquidity and ensuring that users can execute their trading strategies efficiently.

The OrderBook contract in a decentralized finance (DeFi) platform acts as a crucial component managing orders for increasing, decreasing, and swapping positions. It maintains structured records of orders, facilitating their creation, execution, modification, and cancellation with a focus on leveraging price feeds from various sources to ensure accurate and fair trading conditions. Below, we elaborate on these functionalities with Solidity snippets.

### Order Indexing and Structure

Orders are indexed and stored within mappings, allowing efficient access and management. The structure for `IncreaseOrder` and `SwapOrder` includes details such as account, tokens involved, amounts, and conditions for execution.

```solidity

// Mapping of account to orderIndex to Order

mapping (address => mapping(uint256 => SwapOrder)) public swapOrders;

// Mapping of account to index

mapping (address => uint256) public swapOrdersIndex;

```

### Creating an Increase Order

Users can create orders to increase their positions, specifying parameters such as token paths, amounts, leverage, trigger prices, and fees. The function handles the transfer of input tokens and wraps them if necessary.

```solidity

function createIncreaseOrder(

address[] memory \_path,

uint256 \_amountIn,

address \_indexToken,

uint256 \_minOut,

uint256 \_sizeDelta,

address \_collateralToken,

bool \_isLong,

uint256 \_triggerPrice,

bool \_triggerAboveThreshold,

uint256 \_executionFee,

bool \_shouldWrap

) external payable nonReentrant {

// Execution fee payment and token handling logic...

}

```

### Executing an Increase Order

Upon meeting the trigger conditions, an increase order is executed. This involves verifying prices, transferring tokens, and adjusting the position accordingly.

```solidity

function executeIncreaseOrder(

address \_address,

uint256 \_orderIndex,

address payable \_feeReceiver

) external nonReentrant {

// Order fetching, validation, and execution logic...

}

```

### Canceling an Increase Order

Users can cancel their orders if they change their minds or market conditions shift, ensuring flexibility and control over their trading strategies.

```solidity

function cancelIncreaseOrder(uint256 \_orderIndex) public nonReentrant {

// Order cancellation logic...

}

```

### Modifying an Increase Order

This allows users to adjust the parameters of an existing order, such as size and trigger price, adapting to evolving market conditions without needing to cancel and recreate the order.

```solidity

function updateIncreaseOrder(uint256 \_orderIndex, uint256 \_sizeDelta, uint256 \_triggerPrice, bool \_triggerAboveThreshold) external nonReentrant {

// Order update logic...

}

```

### Price Feeds and Index Price

The PriceFeed contract integrates prices from various exchanges like Binance, Bitfinex, and Coinbase, along with Chainlink, to determine accurate market prices. Keepers update these feeds within thresholds to maintain relevance and accuracy. The system uses these feeds to execute orders based on set conditions, including trigger prices for orders and liquidation thresholds for positions.

```solidity

// Price feed integration and keeper logic for accurate and up-to-date pricing...

```

These functionalities highlight the complex interplay between user orders, market conditions, and platform mechanics, ensuring a dynamic and responsive trading environment on DeFi platforms.

The Oracle feed mechanism plays a vital role in decentralized finance (DeFi) platforms, providing real-time price information that is crucial for executing trades, managing positions, and ensuring platform integrity. Below, we delve into the implementation of price feed mechanisms, including how to handle stablecoin pricing and spread adjustments.

### Primary Price Feed

The primary price feed function retrieves the current price of a token, leveraging Chainlink or other reliable data sources to ensure accuracy. It incorporates checks for data freshness and the operational status of Chainlink nodes to guard against stale or manipulated data.

```solidity

function getPrimaryPrice(

address \_token,

bool \_maximise

) public override view returns (uint256) {

// Retrieve the PriceFeed address from the mapping

address priceFeedAddress = priceFeeds[\_token];

require(priceFeedAddress != address(0), "VaultPriceFeed: invalid price feed");

// Check for Chainlink data source availability

if (chainlinkFlags != address(0)) {

bool isRaised = IChainlinkFlags(chainlinkFlags).getFlag(FLAG\_ARBITRUM\_SEQ\_OFFLINE);

if (isRaised) {

// If the flag is raised, critical operations shouldn't be performed

revert("Chainlink feeds are not being updated");

}

}

IPriceFeed priceFeed = IPriceFeed(priceFeedAddress);

uint256 price = 0;

uint80 roundId = priceFeed.latestRound();

// Loop through the price feed to obtain a sample

for (uint80 i = 0; i < priceSampleSpace; i++) {

if (roundId <= i) { break; }

uint256 p;

// Retrieve the latest or historical price

if (i == 0) {

int256 \_p = priceFeed.latestAnswer();

require(\_p > 0, "VaultPriceFeed: invalid price");

p = uint256(\_p);

} else {

(, int256 \_p, , ,) = priceFeed.getRoundData(roundId - i);

require(\_p > 0, "VaultPriceFeed: invalid price");

p = uint256(\_p);

}

// Determine the highest or lowest price based on \_maximise

if (\_maximise && p > price || !\_maximise && p < price) {

price = p;

}

}

require(price > 0, "VaultPriceFeed: could not fetch price");

// Adjust for token decimal precision

uint256 \_priceDecimals = priceDecimals[\_token];

return price.mul(PRICE\_PRECISION).div(10 \*\* \_priceDecimals);

}

```

### Handling Stablecoins

Stablecoins require special handling due to their pegged nature. The system checks if the oracle price deviates significantly from the peg; if it does, adjustments are made to align with expected values.

```solidity

if (strictStableTokens[\_token]) {

uint256 delta = price > ONE\_USD ? price.sub(ONE\_USD) : ONE\_USD.sub(price);

if (delta <= maxStrictPriceDeviation) {

return ONE\_USD;

}

if (\_maximise && price > ONE\_USD || !\_maximise && price < ONE\_USD) {

return price;

}

return ONE\_USD;

}

```

### Spread Adjustment

The spread is adjusted based on whether the aim is to maximise the price or not, considering the token's specific spread basis points.

```solidity

uint256 \_spreadBasisPoints = spreadBasisPoints[\_token];

return \_maximise ?

price.mul(BASIS\_POINTS\_DIVISOR.add(\_spreadBasisPoints)).div(BASIS\_POINTS\_DIVISOR) :

price.mul(BASIS\_POINTS\_DIVISOR.sub(\_spreadBasisPoints)).div(BASIS\_POINTS\_DIVISOR);

```

### Fast Price Feed

Fast price feeds offer an alternative pricing mechanism that reacts more quickly to market movements compared to traditional oracle systems. They are adjusted for token volatility and the reliability of the data source, with specific conditions triggering the use of a spread between the fast price and reference prices.

The implementation details involve handling various scenarios where the fast price may be outdated or deviates significantly from the reference price. Special attention is given to the handling of stablecoins and volatile tokens, ensuring that prices used for trading and position management reflect the current market state as accurately as possible.

This sophisticated approach to price feeds underscores the complexity and innovation inherent in DeFi platform development, highlighting the careful balance between ensuring data accuracy, responding to market conditions, and protecting against potential data manipulation or stale information.

GMX serves as a governance token within its platform, facilitating various functionalities like staking, fee collection, liquidity provision, and more through smart contracts. Below, we'll delve into the implementations for staking, updating rewards, migration, and access control, illustrating how GMX enriches its ecosystem.

### Staking

Staking involves locking tokens to participate in platform governance, earn rewards, or provide liquidity. The `\_stake` function handles the token transfer and updates the staking metrics accordingly.

```solidity

function \_stake(

address \_fundingAccount,

address \_account,

address \_depositToken,

uint256 \_amount

) private {

// Transfer tokens to the contract and update the staking metrics...

}

```

### Updating Rewards

Rewards are distributed to stakers based on their contribution. The `\_updateRewards` function calculates and allocates rewards to users, adjusting for the cumulative reward per token and ensuring fair distribution.

```solidity

function \_updateRewards(address \_account) private {

// Calculate rewards based on staked amounts and update user metrics...

}

```

### Migration

The platform may allow for the migration of assets, typically during upgrades or to introduce new functionalities. The migration process involves users transferring their holdings in return for equivalent or upgraded assets.

```solidity

function migrate(

address \_token,

uint256 \_tokenAmount

) public nonReentrant {

// Validate migration conditions and execute token transfer and minting...

}

```

### Access Control

Dynamic parameters within the platform, such as fees, leverage limits, and token weights within the GLP pool, are adjusted by governance actions to adapt to changing market conditions and strategic objectives.

- \*\*Swap and Position Trading Fees\*\*: Adjusted dynamically by governance, with a cap to protect users.

- \*\*GLP Token Weights\*\*: Influence swap prices, dynamically adjusted to balance the liquidity pool.

- \*\*Pause Swap and Leverage Trading\*\*: A mechanism to safeguard the platform during extreme market conditions.

- \*\*Maximum Leverage and Position Capacity\*\*: Adjusted to manage risk across long and short positions.

Parameters subject to TimeLock adjustments include adding new tokens, updating price feeds, and governance values, ensuring changes are deliberate and transparent.

### GLP Token

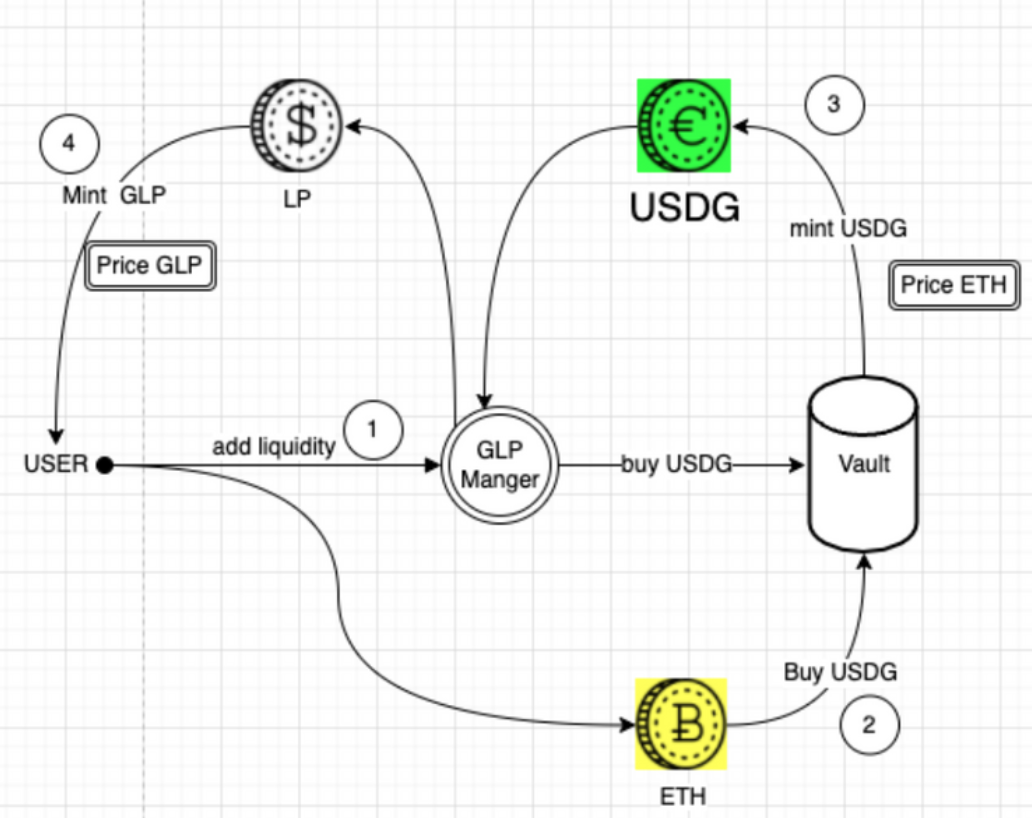
The GLP token represents a share in the liquidity pool, with holders providing liquidity through token purchases that are indexed to the GLP value. The smart contracts manage the liquidity pool's token balances, debt obligations, and collected fees, fostering an efficient and balanced ecosystem.

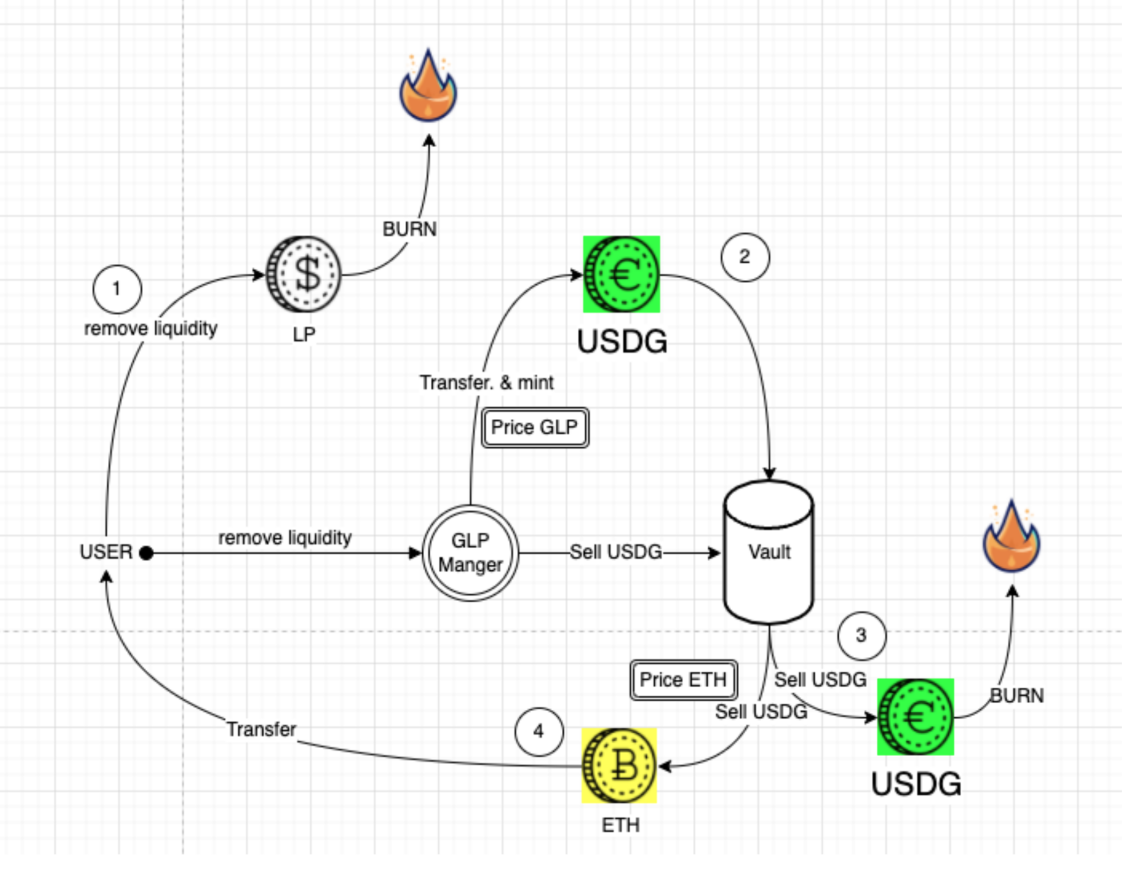
```solidity

// GLP and liquidity provision mechanisms...

```

These smart contract snippets illustrate the multifaceted functionalities within the GMX ecosystem, showcasing the platform's ability to manage governance, rewards, liquidity, and migration securely and efficiently. Through these mechanisms, GMX aims to create a robust and flexible DeFi environment that adapts to the evolving needs and contributions of its users.

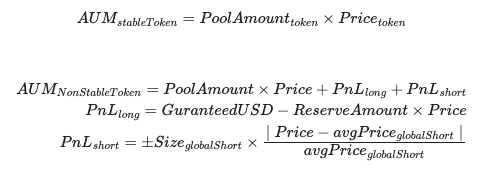




GLP's pricing not only depends on the price of its constituent assets but also considers the profit and loss situation of GLP in the game against traders. Since GLP and traders are in a zero-sum game where traders' losses translate to GLP gains (thus increasing its token price) and vice versa, the minting and redemption price of GLP is calculated based on the total value of assets in the index, including the profit and loss of open positions, divided by the GLP supply.

GLP holders provide liquidity for leveraged trades, earning 70% of the funding rate fees. The mint, burn, and swap prices of GLP will fluctuate based on the impact of these actions on asset prices. Token weights are adjusted based on open contracts to hedge the risks for GLP holders, potentially creating arbitrage opportunities. For instance, if ETH is heavily traded long, its weight will increase, thus affecting the price of GLP and related assets. Token weights are adjusted based on traders' open positions to help hedge GLP holders.

Calculating AUM involves considering each asset's poolAmount and the PnL of positions. The Solidity code for getAum demonstrates how to calculate AUM by summing up the values of whitelisted tokens, adjusting for stablecoins, and accounting for the PnL from short and long positions. This calculation ensures a fair valuation of the GLP index based on current market conditions and the platform's trading activity.



### Adding Liquidity

The execution of positions is subject to a timelock, influencing the liquidity provision process.

```solidity

function \_addLiquidity(

address \_fundingAccount,

address \_account,

address \_token,

uint256 \_amount,

uint256 \_minUsdg,

uint256 \_minGlp

) private returns (uint256) {

require(\_amount > 0, "GlpManager: invalid \_amount");

// Token transfer and accounting

IERC20(\_token).safeTransferFrom(\_fundingAccount, address(vault), \_amount);

uint256 usdgAmount = vault.buyUSDG(\_token, address(this));

require(usdgAmount >= \_minUsdg, "GlpManager: insufficient USDG output");

// Mint GLP

uint256 mintAmount = /\* Calculation based on AUM and GLP supply \*/;

IMintable(glp).mint(\_account, mintAmount);

// Update state

lastAddedAt[\_account] = block.timestamp;

return mintAmount;

}

```

### Calculating AUM Value

AUM calculation considers each asset's `poolAmount` and the PnL of positions:

```solidity

function getAum(

bool maximise

) public view returns (uint256) {

uint256 aum = aumAddition; // Initial AUM value set by system admin

uint256 shortProfits = 0;

IVault \_vault = vault;

for (uint256 i = 0; i < vault.allWhitelistedTokensLength(); i++) {

address token = vault.allWhitelistedTokens(i);

if (!vault.whitelistedTokens(token)) {

continue;

}

uint256 price = maximise ? \_vault.getMaxPrice(token) : \_vault.getMinPrice(token);

uint256 poolAmount = \_vault.poolAmounts(token);

uint256 decimals = \_vault.tokenDecimals(token);

if (\_vault.stableTokens(token)) {

aum += poolAmount.mul(price).div(10 \*\* decimals);

} else {

// Calculations for non-stable tokens including short and long positions' impacts on AUM

uint256 size = \_vault.globalShortSizes(token);

if (size > 0) {

(uint256 delta, bool hasProfit) = getGlobalShortDelta(token, price, size);

if (!hasProfit) {

aum += delta;

} else {

shortProfits += delta;

}

}

aum += \_vault.guaranteedUsd(token);

uint256 reservedAmount = \_vault.reservedAmounts(token);

aum += poolAmount.sub(reservedAmount).mul(price).div(10 \*\* decimals);

}

}

aum = shortProfits > aum ? 0 : aum.sub(shortProfits);

return aumDeduction > aum ? 0 : aum.sub(aumDeduction);

}

```

This system for pricing GLP and managing liquidity provision showcases a sophisticated balance between rewarding liquidity providers and maintaining a competitive and fair trading environment. Adjusting GLP's price and supply in response to market conditions and traders' actions ensures the platform can sustainably support leveraged trading activities.

To mitigate the risk of flash loan attacks, the GMX platform incorporates several key mechanisms into its design, particularly concerning GLP activities like opening and closing positions, swapping, and liquidity provision (LP) actions such as depositing and redeeming. Here's a closer look at these protective measures:

### Use of Oracle Prices and TimeLock

By utilizing oracle prices for critical operations like opening/closing GLP positions, swapping, and LP transactions, GMX ensures that these actions are based on reliable and timely price information. The introduction of a TimeLock mechanism further enhances security by preventing immediate, arbitrary changes that could be exploited in flash loan attacks. TimeLock delays the execution of sensitive operations, providing a window to detect and respond to potential manipulative activities.

### Limitations on LP Actions

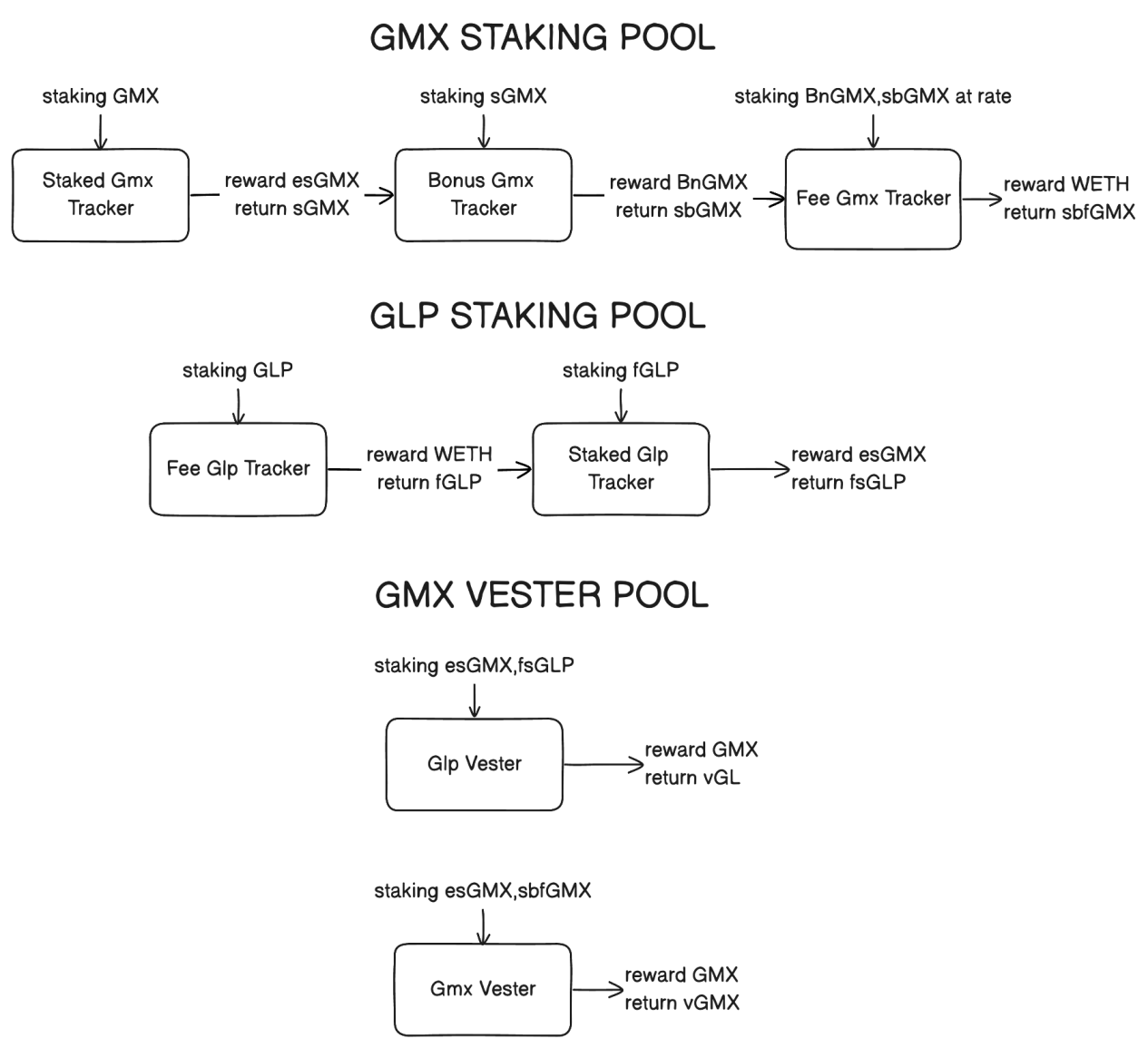
To prevent liquidity providers from rapidly adding or removing liquidity—which could facilitate flash loan attacks—GMX imposes restrictions on the frequency of these actions. By requiring a minimum duration between liquidity transactions, the platform ensures that LPs cannot engage in behaviors that might destabilize the market or facilitate attacks.

### Position Holding Requirements for Traders

On the trader side, particularly in Contract For Difference (CFD) trading, GMX mandates that positions must be held for a minimum duration before PnL can be calculated. This requirement prevents traders from opening and closing positions within a very short timeframe to exploit market movements artificially, reducing the risk of market manipulation and flash loan attacks.

### Staking Flow Improvements

GMX has refined its staking mechanism, particularly in its implementation on the Binance Smart Chain (BSC), by maintaining the core logic of a MasterChef-style staking system. In interactions with multiple tokens, the primary design philosophy revolves around staking, custody, and vesting. This approach ensures that the staking process is secure and that rewards are distributed fairly and transparently over time, reducing the risk of manipulation.



These measures collectively strengthen the GMX platform against potential flash loan attacks and other forms of market manipulation, ensuring a more stable and secure trading environment for all participants. By carefully balancing the flexibility and openness characteristic of DeFi platforms with robust security mechanisms, GMX aims to provide a trustworthy and efficient platform for trading and liquidity provision.