# Overlay V1 Core Module System

The module system has two key components:

1. Collaterals Module
2. Markets Module

## Collaterals Module

Collaterals Module consists of collateral managers specializing in different types of collateral. Trader interactions with the system occur through collateral managers. Collateral managers are given mint and burn permissions on the OVL token by the mothership contract.

Each manager has external functions:

- `build()`
- `unwind()`
- `liquidate()`

Currently, we have an OVL Collateral Manager that accepts OVL: collateral/OverlayV1OVLCollateral.sol

#### OverlayV1OVLCollateral.sol:

`build(address _market, uint256 _collateral, uint256 _leverage, bool _isLong):`

- Auth calls `IOverlayV1Market(_market).enterOI()` which queues open interest on the market contract, adjusted for trading and impact fees
- Transfers OVL collateral amount to manager from `msg.sender`
- Returns ERC1155 position token for user's share of the position

`unwind(uint256 _positionId, uint256 _shares):`

- Auth calls `IOverlayV1Market(_market).exitData()` view which returns open interest occupied by position & change in price since entry
- Calculates current value less fees of position being unwound given ERC1155 `_shares`
- Mints `PnL = value - cost` in OVL to collateral manager if PnL > 0 or burns if PnL < 0 from collateral manager
- Transfers value to `msg.sender`
- Auth calls `IOverlayV1Market(_market).exitOI()` which removes open interest from market contract
- Burns ERC1155 position token shares

`liquidate(uint256 _positionId):`

- Auth calls `IOverlayV1Market(_market).exitData()` view which returns open interest occupied by position & change in price since entry
- Checks if position value is less than initial open interest times maintenance margin
- Auth calls `IOverlayV1Market(_market).exitOI()` which removes open interest from market contract
- Zeroes the position's share of total open interest on long or short side
- Burns `loss = cost - value` in OVL from collateral manager
- Transfers reward to liquidator

## Markets Module

Markets module consists of markets on different data streams.

Each market tracks:

- Total open interest outstanding on long and short sides: `OverlayV1OI.__oiLong__` and `OverlayV1OI.__oiShort__`
- Accumulator snapshots for how much of the open interest cap has been entered into: `OverlayV1Comptroller.impactRollers`
- Accumulator snapshots for how much OVL has been printed: `OverlayV1Comptroller.brrrrdRollers`
- Historical prices fetched from the oracle: `OverlayV1PricePoint._pricePoints`
- Collateral managers approved by governance to add/remove open interest: `OverlayV1Governance.isCollateral`

Each market has external functions accessible only by approved collateral managers:

- `enterOI()`
- `exitData()`
- `exitOI()`

and an external `update()` function to be called in the event the market hasn't been interacted with for an extended period of time.

Currently, we have Overlay markets on Uniswap V3 oracles: OverlayV1UniswapV3Market.sol which implements markets/OverlayV1Market.sol

#### OverlayV1Market.sol:

`enterOI(bool _isLong, uint256 _collateral, uint256 _leverage):`

- Internal calls `OverlayV1UniswapV3Market.entryUpdate()` which fetches and stores a new price from the oracle and applies funding to the open interest
- Internal calls `OverlayV1Comptroller.intake()` which calculates and records the market impact
- Internal calls `OverlayV1OI.queueOi()` to add the adjusted open interest to the market

`exitData(bool _isLong, uint256 _pricePoint, uint256 _compounding):`

- Internal calls `OverlayV1UniswapV3Market.exitUpdate()` which fetches current and last settlement prices from the oracle and applies funding
- Returns total open interest on side of trade and ratio between exit and entry prices

`exitOI(bool _isLong, bool _fromQueued, uint _oi, uint _oiShares, uint _brrrr, uint _antiBrrrr):`

- Internal calls `OverlayV1Comptroller.brrrr()` which records the amount of OVL minted or burned for trade
- Removes open interest from the long or short side

`update():`

- Internal calls `OverlayV1UniswapV3Market.staticUpdate()` to update the market

### OverlayV1Comptroller.sol:

`intake(bool _isLong, uint _oi):`

- Records in accumulator snapshots `impactRollers` the amount of open interest cap occupied by the trade: `oi / oiCap()`
- Calculates market impact fee `_oi * (1 - e**(-lmbda * (impactRollers[now] - impactRollers[now-impactWindow])))` in OVL burned from collateral manager
- Internal calls `brrrr()` to record the impact fee that will be burned

`brrrr(uint _brrrr, _antiBrrrr):`

- Records in accumulator snapshots `brrrrdRollers` an amount of OVL minted `_brrrr` or burned `_antiBrrrr`

`oiCap():`

- Returns the current dynamic cap on open interest for the market, if less than constraint from `OverlayV1UniswapV3Market.depth()`: `staticCap * min(1, 2 - (brrrrdRollers[now] - brrrrdRollers[now-brrrrdWindowMacro]) / brrrrdExpected)`

### OverlayV1OI.sol:

`updateFunding(uint _epochs):`

- Internal calls `payFunding()` which pays funding between `__oiLong__` and `__oiShort__`: open interest imbalance is drawn down by `(1-2*k)**(epochs)`
- Internal calls `updateOi()` which transfers queued open interest into `__oiLong__` and `__oiShort__` since now eligible for funding

`queueOi(bool _isLong, uint256 _oi, uint256 _oiCap):`

- Add open interest to either `__queuedOiLong__` or `__queuedOiShort__`
- Checks current open interest cap has not been exceeded: `_oiLong__ + __queuedOiLong__ <= _oiCap` or `_oiShort__ + __queuedOiShort__ <= _oiCap`

### OverlayV1PricePoint.sol:

`setPricePointCurrent(PricePoint memory _pricePoint):`

- Stores a new historical price in the `_pricePoints` array. Price points include bid and ask values used for entry and exit: `PricePoint{ uint bid; uint ask; uint price }`. Longs receive the ask on entry, bid on exit. Shorts receive the bid on entry, ask on exit.

`insertSpread(uint _microPrice, uint _macroPrice)`

- Calculates bid and ask values given shorter and longer TWAP values fetched from the oracle
- Applies the static spread `pbnj` to bid `e**(-pbnj)` and ask `e**(pbnj)`

### OverlayV1UniswapV3Market.sol:

`price(uint _ago):`

- External calls `IUniswapV3Pool(marketFeed).observe()` for tick cumulative snapshots from `_ago`, `_ago+microWindow`, and `_ago+macroWindow` seconds ago
- Calculates TWAP values for both the `macroWindow` and `microWindow` window sizes
- Returns a new price point through internal call to `OverlayV1PricePoint.insertSpread()`

`depth():`

- External calls `IUniswapV3Pool(marketFeed).observe()` for `secondsPerLiquidityCumulativeX128` snapshots from now and `microWindow` seconds ago to calculate amount of virtual ETH reserves in Uniswap V3 pool: `_ethAmount`
- External calls `IUniswapV3Pool(ovlFeed).observe()` for `tickCumulative` snapshots from now and `microWindow` seconds ago to calculate current OVL price relative to ETH: `_price`
- Returns bound on open interest cap from virtual liquidity in Uniswap pool: `(lmbda * _ethAmount / _price) / 2`

`entryUpdate():`

- Internal calls `price()` to fetch a new price point if at least one `updatePeriod` has passed since the last fetch
- Internal calls `OverlayV1PricePoint.setPricePointCurrent()` to store fetched price
- Internal calls `updateFunding()` if at least one `compoundingPeriod` has passed since the last funding

`exitUpdate():`

- Internal calls `price()` to fetch a new price point for the last position built, `entryPrice`, if at least one `updatePeriod` has passed since the last fetch
- Internal calls `OverlayV1PricePoint.setPricePointCurrent()` to store the fetched price for last position built
- Internal calls `price()` again to fetch the latest price point for an `exitPrice`, if more than one `updatePeriod` has passed
- Internal calls `OverlayV1PricePoint.setPricePointCurrent()` again to store the fetched price for exit
- Internal calls `updateFunding()` if at least one `compoundingPeriod` has passed since the last funding

`staticUpdate():`

- Internal calls `price()` to fetch a new price point if at least one `updatePeriod` has passed since the last fetch
- Internal calls `OverlayV1PricePoint.setPricePointCurrent()` to store fetched price
- Internal calls `updateFunding()` if at least one `compoundingPeriod` has passed since the last funding
- Needed to update the market in the event no recent trading activity has occurred, since Uniswap V3 pools only store a limited number of historical snapshots for the tick and liquidity oracle

## Nuances:

Queued open interest:

- Queued open interest ( `__queuedOiLong__` , `__queuedOiShort__` ) is open interest that is not yet eligible for funding. It is transferred over to ( `__oiLong__` , `__oiShort__` ) after the last `compoundingPeriod` has passed through an internal call to `updateOi()`

Price updates:

- Positions settle at the price that occurs one `updatePeriod` after the block in which the position was built: what we call `t+1` . This is to prevent front-running within the update period
- Positions exit, however, at the last price available from the oracle. As there isn't a front-running issue on exit.
- `OverlayV1UniswapV3Market.entryUpdate()` fetches the price for previously built positions one `updatePeriod` after they were built using historical tick cumulative snapshots from Uniswap V3
- `OverlayV1UniswapV3Market.exitUpdate()` also needs the latest price for the position exiting. Performs a double fetch if more than one update period has passed: 1. Fetches price for the last position built several update periods ago; 2. Fetches latest price for position exiting at current time.