Archimedes

Auto-generated by code2pdf

March 22, 2023

Contents

References index

A

A

A

A

B

C

ParameterStore.changeCurveMaxExchangeGuard(**newCurveMaxExchangeGuard**) X
onlyGovernor, 80

C

D

_dai
write
Zapper.setDependencies(addressLevEngine, addressArchToken, addressParamStore) X nonReentrant onlyAdmin, 123
definition
Zapper._dai: IERC20Upgradeable, 115
read
Zapper.setDependencies(addressLevEngine, addressArchToken, addressParamStore) X nonReentrant onlyAdmin, 123

deletePosition
call
Coordinator.unwindLeveragedOUSD(_nftId, _userAddress) X nonReentrant onlyExecutive, 128
definition
CDPosition.deletePosition(nftID) X nftIDMustExist canDeletePosition nonReentrant onlyExecutive, 68

deposit
call
VaultOUSD.archimedesDeposit(assets, receiver) X nonReentrant onlyExecutive, 134
definition
VaultOUSD.deposit(assets, receiver) X, 134

depositCollateralUnderNFT
call
LeverageEngine._createLeveragedPosition(ousdPrinciple, cycles, maxArchAmount, userAddress, minLeverageAmount), 111
definition
Coordinator.depositCollateralUnderNFT(_nftId, _amountInOUSD) X [ICoordinator], 131
Coordinator.depositCollateralUnderNFT(_nftId, _amountInOUSD) X nonReentrant onlyExecutive, 127

depositOUSDtoPosition
call
Coordinator.getLeveragedOUSD(_nftId, _amountToLeverage) X nonReentrant onlyExecutive, 127
definition
CDPosition.depositOUSDtoPosition(nftID, oUSDAmountToDeposit) X nftIDMustExist nonReentrant onlyExecutive, 70

_disableInitializers
call
CDPosition.constructor() X, 72
PositionToken.constructor() X, 90
Exchanger.constructor() X, 94
LeverageEngine.constructor() X, 112
Coordinator.constructor() X, 129
Auction.constructor() X, 107
ParameterStore.constructor() X, 78
VaultOUSD.constructor() X, 135

E

F

F

G

G

H

I

L

M

N

O

P

R

S

T

V

W

x

Chapter 1

ArchToken

1.1 contract ArchToken

```
/**
@title Archimedes Governance token
@notice contract is ERC20Permit and ERC20Votes to allow voting
 **/

contract ArchToken is ERC20, BasicAccessController, ERC20Permit, ERC20Votes {
}
```

1.1 constructor(_addressTreasury) X

```
    constructor(address _addressTreasury) ERC20("ARCH", "ARCH") ERC20Permit("ArchToken") {
        _mint(_addressTreasury, 100000000 ether);
        _grantRole(ADMIN_ROLE, _msgSender());
    }
```

1.1 _afterTokenTransfer(from, to, amount)

```
    // The following functions are overrides required by Solidity.
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal override(ERC20, ERC20Votes) {
        super._afterTokenTransfer(from, to, amount);
    }
```

1.1 _mint(to, amount)

```
    function _mint(address to, uint256 amount) internal override(ERC20, ERC20Votes) {
        super._mint(to, amount);
    }
```

1.1 _burn(account, amount)

```
    function _burn(address account, uint256 amount) internal override(ERC20, ERC20Votes) {
        super._burn(account, amount);
    }
```

Chapter 2

CDPosition

2.1 contract *CDPosition*

```
/// @title CDPosition is ledger contract for all  NFT positions and regular positions
/// @dev CDP creates and destroy NFT and address positions. It keep tracks of how many tokens
    ↪  user has borrowed.
/// It keeps track of how much interest each position accrue
/// @notice CDPosition does not hold any tokens. It is not a vault of any kind.
/// @notice CDP does not emit any events. All related events will be emitted by the calling
    ↪ contract.
/// @notice This contract (will be) proxy upgradable
contract CDPosition is AccessController, UUPSUpgradeable, ReentrancyGuardUpgradeable {

    mapping(uint256 => CDP) internal _nftCDP;

    address internal _addressVaultOUSD;
    address internal _addressParameterStore;
    VaultOUSD internal _vault;
    ParameterStore internal _parameterStore;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

2.1 struct *CDPosition*.CDP

```
    struct CDP {
        uint256 oUSDPrinciple; // Amount of OUSD originally deposited by user
        uint256 oUSDTotalWithoutInterest; // Principle + OUSD acquired from selling borrowed
            ↪ lvUSD
        uint256 lvUSDBorrowed; // Total lvUSD borrowed under this position
        uint256 shares; // Total vault shares allocated to this position
        // // New values, need to implement changing values
        uint256 openTimeStamp; // Open time
        uint256 positionLifetimeInDays; // Position in days
        uint256 positionExpiration;
    }
```

2.1 modifier ***nftIDMustExist***(nftID)

```
    // Maps return default value when entry is not present. OUSD principle will always be gt
        ↪ 0 if _nftCDP has
    // a valid value in nftID
    modifier nftIDMustExist(uint256 nftID) {
        require(_nftCDP[nftID].oUSDPrinciple != 0, "NFT ID must exist");
        _;
    }
```

2.1 modifier nftIDMustNotExist(**nftID**)

```
    modifier nftIDMustNotExist(uint256 nftID) {
        require(_nftCDP[nftID].oUSDPrinciple == 0, "NFT ID must not exist");
        _;
    }
```

2.1 modifier **canDeletePosition**(**nftID**)

```
    modifier canDeletePosition(uint256 nftID) {
        require(_nftCDP[nftID].lvUSDBorrowed == 0, "lvUSD borrowed must be zero");
        _;
    }
```

2.1 **createPosition**(**nftID**, **oOUSDPrinciple**) X nftIDMustNotExist *nonReentrant* *onlyExecutive*

```
    /// @dev add new entry to nftid<>CPP map with ousdPrinciple.
    /// Update both principle and total with OUSDPrinciple
    /// @param nftID newly minted NFT
    /// @param oOUSDPrinciple initial OUSD investment (ie position principle)
    function createPosition(uint256 nftID, uint256 oOUSDPrinciple) external
        ↪ nftIDMustNotExist(nftID) nonReentrant onlyExecutive {
        uint256 blockTimestamp = block.timestamp;
        uint256 positionTimeToLive = _parameterStore.getPositionTimeToLiveInDays();
        uint256 positionEndDate = blockTimestamp + positionTimeToLive * 1 days;
        _nftCDP[nftID] = CDP(oOUSDPrinciple, oOUSDPrinciple, 0, 0, blockTimestamp,
            ↪ positionTimeToLive, positionEndDate);
    }
```

2.1 ***deletePosition***(**nftID**) X ***nftIDMustExist*** **canDeletePosition** *nonReentrant* *onlyExecutive*

```
    /// @dev delete entry in CDP --if-- lvUSD borrowed balance is zero
    ///
    /// @param nftID entry address to delete
    function deletePosition(uint256 nftID) external nftIDMustExist(nftID) canDeletePosition(
        ↪ nftID) nonReentrant onlyExecutive {
        /// Set all values to default. Not way to remove key from mapping in solidity
        delete _nftCDP[nftID];
    }
```

2.1 addSharesToPosition(nftID, shares) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev add shares to a position.
    /// @param nftID NFT position to update
    /// @param shares shares to add
    function addSharesToPosition(uint256 nftID, uint256 shares) external nftIDMustExist(
        ↪ nftID) nonReentrant onlyExecutive {
        _nftCDP[nftID].shares += shares;
    }
```

2.1 removeSharesFromPosition(nftID, shares) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev remove shares from position.
    /// @param nftID NFT position to update
    /// @param shares shares to remove
    function removeSharesFromPosition(uint256 nftID, uint256 shares) external nftIDMustExist(
        ↪ nftID) nonReentrant onlyExecutive {
        require(_nftCDP[nftID].shares >= shares, "Shares exceed position balance");
        _nftCDP[nftID].shares -= shares;
    }
```

2.1 borrowLvUSDFromPosition(nftID, lvUSDAmountToBorrow) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev update borrowed lvUSD in position. This method adds a delta to existing borrowed
        ↪  value
    /// @param nftID NFT position to update
    /// @param lvUSDAmountToBorrow amount to add to position's existing borrowed lvUSD sum
    function borrowLvUSDFromPosition(uint256 nftID, uint256 lvUSDAmountToBorrow) external
        ↪ nftIDMustExist(nftID) nonReentrant onlyExecutive {
        _nftCDP[nftID].lvUSDBorrowed += lvUSDAmountToBorrow;
    }
```

2.1 repayLvUSDToPosition(nftID, lvUSDAmountToRepay) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev update borrowed lvUSD in position. This method removed a delta to existing
        ↪ borrowed value
    /// @param nftID NFT position to update
    /// @param lvUSDAmountToRepay amount to remove fom position's existing borrowed lvUSD sum
    function repayLvUSDToPosition(uint256 nftID, uint256 lvUSDAmountToRepay) external
        ↪ nftIDMustExist(nftID) nonReentrant onlyExecutive {
        if (_nftCDP[nftID].lvUSDBorrowed < lvUSDAmountToRepay) {
            // if trying to repay more lvUSDthen expected, zero out lvUSDBorrowed
            _nftCDP[nftID].lvUSDBorrowed = 0;
        } else {
            _nftCDP[nftID].lvUSDBorrowed -= lvUSDAmountToRepay;
        }
    }
```

2.1 depositOUSDtoPosition(nftID, oUSDAmountToDeposit) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev update deposited OUSD in position. This method adds a delta to existing
        ↪ deposited value
    /// @param nftID NFT position to update
    /// @param oUSDAmountToDeposit amount to add to position's existing deposited sum
    function depositOUSDtoPosition(uint256 nftID, uint256 oUSDAmountToDeposit) external
        ↪ nftIDMustExist(nftID) nonReentrant onlyExecutive {
        _nftCDP[nftID].oUSDTotalWithoutInterest += oUSDAmountToDeposit;
    }
```

2.1 withdrawOUSDFromPosition(nftID, oUSDAmountToWithdraw) X nftIDMustExist nonReentrant onlyExecutive

```
    /// @dev update deposited OUSD in position. This method removed a delta to existing
        ↪ deposited value
    /// @param nftID NFT position to update
    /// @param oUSDAmountToWithdraw amount to remove to position's existing deposited sum
    function withdrawOUSDFromPosition(uint256 nftID, uint256 oUSDAmountToWithdraw) external
        ↪ nftIDMustExist(nftID) nonReentrant onlyExecutive {
        require(getOUSDTotalIncludeInterest(nftID) >= oUSDAmountToWithdraw, "Insufficient
            ↪ OUSD balance");
        _nftCDP[nftID].oUSDTotalWithoutInterest -= oUSDAmountToWithdraw;
    }
```

2.1 getOUSDPrinciple(nftID) nftIDMustExist

```
    // * CDP Getters *//
    function getOUSDPrinciple(uint256 nftID) external view nftIDMustExist(nftID) returns (
        ↪ uint256) {
        return _nftCDP[nftID].oUSDPrinciple;
    }
```

2.1 getOUSDInterestEarned(nftID) nftIDMustExist

```
    function getOUSDInterestEarned(uint256 nftID) public view nftIDMustExist(nftID) returns (
        ↪ uint256) {
        uint256 sharesOfOwner = _nftCDP[nftID].shares;
        uint256 totalFundsFromPreviewRedeem = _vault.previewRedeem(sharesOfOwner);
        uint256 outTotal = _nftCDP[nftID].oUSDTotalWithoutInterest;
        if (_nftCDP[nftID].oUSDTotalWithoutInterest > totalFundsFromPreviewRedeem) {
            // revert("InterestEarned calc error");
            return 0;
        }
        return totalFundsFromPreviewRedeem - _nftCDP[nftID].oUSDTotalWithoutInterest;
    }
```

2.1 **getOUSDTotalIncludeInterest**(nftID) **nftIDMustExist**

```
    function getOUSDTotalIncludeInterest(uint256 nftID) public view nftIDMustExist(nftID)
        ↪ returns (uint256) {
        return _nftCDP[nftID].oUSDTotalWithoutInterest + getOUSDInterestEarned(nftID);
    }
```

2.1 getOUSDTotalWithoutInterest(nftID) **nftIDMustExist**

```
    function getOUSDTotalWithoutInterest(uint256 nftID) external view nftIDMustExist(nftID)
        ↪ returns (uint256) {
        return _nftCDP[nftID].oUSDTotalWithoutInterest;
    }
```

2.1 getLvUSDBorrowed(nftID) **nftIDMustExist**

```
    function getLvUSDBorrowed(uint256 nftID) external view nftIDMustExist(nftID) returns (
        ↪ uint256) {
        return _nftCDP[nftID].lvUSDBorrowed;
    }
```

2.1 **getShares**(nftID) **nftIDMustExist**

```
    function getShares(uint256 nftID) external view nftIDMustExist(nftID) returns (uint256) {
        return _nftCDP[nftID].shares;
    }
```

2.1 getPositionTimeOpened(nftID) **nftIDMustExist**

```
    function getPositionTimeOpened(uint256 nftID) external view nftIDMustExist(nftID) returns
        ↪  (uint256) {
        return _nftCDP[nftID].openTimeStamp; // Open time
    }
```

2.1 **getPositionTimeToLive**(nftID) **nftIDMustExist**

```
    function getPositionTimeToLive(uint256 nftID) external view nftIDMustExist(nftID) returns
        ↪  (uint256) {
        return _nftCDP[nftID].positionLifetimeInDays; // Position in days
    }
```

2.1 **getPositionExpireTime**(nftID) **nftIDMustExist**

```
    function getPositionExpireTime(uint256 nftID) external view nftIDMustExist(nftID) returns
        ↪  (uint256) {
        return _nftCDP[nftID].positionExpiration;
    }
```

2.1 constructor() X

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

2.1 **initialize**() X initializer

```
function initialize() public initializer {
    __AccessControl_init();
    __ReentrancyGuard_init();
    __UUPSUpgradeable_init();

    _grantRole(ADMIN_ROLE, _msgSender());
    setGovernor(_msgSender());
    setExecutive(_msgSender());
    setGuardian(_msgSender());
}
```

2.1 **setDependencies**(*addressVaultOUSD*, ***addressParameterStore***) X *nonReentrant* ***onlyAdmin***

```
function setDependencies(address addressVaultOUSD, address addressParameterStore)
    ↪ external nonReentrant onlyAdmin {
    require(addressVaultOUSD != address(0), "cant set to 0 A");
    require(addressParameterStore != address(0), "cant set to 0 A");
    _addressVaultOUSD = addressVaultOUSD;
    _addressParameterStore = addressParameterStore;
    _vault = VaultOUSD(_addressVaultOUSD);
    _parameterStore = ParameterStore(addressParameterStore);
}
```

2.1 **_authorizeUpgrade**(**newImplementation**)

```
// solhint-disable-next-line
function _authorizeUpgrade(address newImplementation) internal override {
    _requireAdmin();
}
```

Chapter 3

LvUSDToken

3.1 contract LvUSDToken

```
/// @title lvUSD token
/// @dev This is the contract for the Archimedes lvUSD USD pegged stablecoin
contract LvUSDToken is ERC20("lvUSD", "lvUSD"), BasicAccessController, ERC20Burnable {
    address internal _mintingDestination = address(0);
}
```

3.1 constructor() X

```
    constructor() {
        _grantRole(ADMIN_ROLE, _msgSender());
        setMinter(_msgSender());
    }
```

3.1 mint(amount) X onlyMinter

```
    /// @dev Mints tokens to a recipient.
    ///
    /// This function reverts if the caller does not have the minter role.
    ///
    /// @param amount the amount of tokens to mint.
    function mint(uint256 amount) external onlyMinter {
        // Only minter can mint
        _mint(_mintingDestination, amount);
    }
```

3.1 setMintDestination(mintDestination) X onlyAdmin

```
    /// @dev change mint address
    /// @param mintDestination new mint destination
    function setMintDestination(address mintDestination) external onlyAdmin {
        _mintingDestination = mintDestination;
    }
```

3.1 contract *ERC20Burnable*

```
/**
 * @dev Extension of {ERC20} that allows token holders to destroy both their own
 * tokens and those that they have an allowance for, in a way that can be
 * recognized off-chain (via event analysis).
 */
abstract contract ERC20Burnable is Context, ERC20 {
}
```

3.1 *burn*(amount) X [*ERC20Burnable*]

```
    /**
     * @dev Destroys `amount` tokens from the caller.
     *
     * See {ERC20-_burn}.
     */
    function burn(uint256 amount) public virtual {
        _burn(_msgSender(), amount);
    }
```

3.1 **burnFrom**(account, amount) X [*ERC20Burnable*]

```
    /**
     * @dev Destroys `amount` tokens from `account`, deducting from the caller's
     * allowance.
     *
     * See {ERC20-_burn} and {ERC20-allowance}.
     *
     * Requirements:
     *
     * - the caller must have allowance for ``accounts``'s tokens of at least
     * `amount`.
     */
    function burnFrom(address account, uint256 amount) public virtual {
        _spendAllowance(account, _msgSender(), amount);
        _burn(account, amount);
    }
```

3.1 contract *Context*

```
/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
}
```

3.1 _msgSender() [Context]

```
function _msgSender() internal view virtual returns (address) {
    return msg.sender;
}
```

3.1 _msgData() [Context]

```
function _msgData() internal view virtual returns (bytes calldata) {
    return msg.data;
}
```

Chapter 4

ParameterStore

4.1 contract *ParameterStore*

```
/// @title ParameterStore is a contract for storing global parameters that can be modified by
    ↪  a privileged role
/// @notice This contract (will be) proxy upgradable
contract ParameterStore is AccessController, UUPSUpgradeable {
    address internal _addressCoordinator;
    address internal _addressExchanger;

    IAuction internal _auction;

    uint256 internal _maxNumberOfCycles; // regular natural number
    uint256 internal _originationFeeRate; // in ether percentage (see initialize for examples
        ↪ )
    uint256 internal _globalCollateralRate; // in percentage
    uint256 internal _rebaseFeeRate; // in ether percentage (see initialize for examples)
    address internal _treasuryAddress;
    uint256 internal _curveGuardPercentage; // in regular (0-100) percentages
    uint256 internal _slippage; // in regular (0-100) percentages
    // maximum allowed "extra" tokens when exchanging
    uint256 internal _curveMaxExchangeGuard;
    uint256 internal _minPositionCollateral;
    uint256 internal _positionTimeToLiveInDays;
    uint256 internal _coordinatorLeverageBalance;

     /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;

    event ParameterChange(string indexed _name, uint256 _newValue, uint256 _oldValue);
    event TreasuryChange(address indexed _newValue, address indexed _oldValue);
}
```

4.1 modifier *onlyInternalContracts*()

```
    modifier onlyInternalContracts() {
        require(msg.sender == _addressCoordinator || msg.sender == _addressExchanger, "Caller
            ↪  is not internal contract");
        _;
    }
```

4.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

4.1 ***initialize***() X initializer

```
    function initialize() public initializer {
        __AccessControl_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());

        _maxNumberOfCycles = 10;
        _originationFeeRate = 5 ether / 1000; // meaning 0.5%
        _globalCollateralRate = 95;
        _rebaseFeeRate = 30 ether / 100; // meaning 30%
        _curveGuardPercentage = 96;
        _slippage = 1; // 1%;
        _curveMaxExchangeGuard = 50; // meaning we allow exchange with get 50% more then we
            ↪ expected
        _minPositionCollateral = 2 ether;
        _positionTimeToLiveInDays = 370;
        _coordinatorLeverageBalance = 0;

        _treasuryAddress = address(0);
        _addressCoordinator = address(0);
        _addressExchanger = address(0);
    }
```

4.1 **setDependencies**(*addressCoordinator*, ***addressExchanger***, addressAuction) X ***onlyAdmin***

```
    function setDependencies(
        address addressCoordinator,
        address addressExchanger,
        address addressAuction
    ) external onlyAdmin {
        require(addressCoordinator != address(0), "cant set to 0 A");
        require(addressExchanger != address(0), "cant set to 0 A");
        require(addressAuction != address(0), "cant set to 0 A");

        _addressCoordinator = addressCoordinator;
        _addressExchanger = addressExchanger;
        _auction = IAuction(addressAuction);
    }
```

4.1 **changeCoordinatorLeverageBalance**(**newCoordinatorLeverageBalance**) X *onlyInternalContracts*

```
    /* Privileged functions */

    function changeCoordinatorLeverageBalance(uint256 newCoordinatorLeverageBalance) external
        ↪  onlyInternalContracts {
        // No checks that I can think of. Seems convoluted to add a check for lvUSD balance
            ↪ as we "trust" internal contracts to check lvUSD
        // balance when needed.
        _coordinatorLeverageBalance = newCoordinatorLeverageBalance;
    }
```

4.1 changeCurveGuardPercentage(newCurveGuardPercentage) X onlyGovernor

```
    function changeCurveGuardPercentage(uint256 newCurveGuardPercentage) external
        ↪ onlyGovernor {
        // curveGuardPercentage must be a number between 80 and 100
        require(newCurveGuardPercentage >= 80 && newCurveGuardPercentage <= 100, "New CGP out
            ↪  of range");
        emit ParameterChange("curveGuardPercentage", newCurveGuardPercentage,
            ↪ _curveGuardPercentage);
        _curveGuardPercentage = newCurveGuardPercentage;
    }
```

4.1 changeSlippage(*newSlippage*) X onlyGovernor

```
    function changeSlippage(uint256 newSlippage) external onlyGovernor {
        // slippage must be a number between 0 and 5
        require(newSlippage != 0 && newSlippage < 5, "New slippage out of range");
        emit ParameterChange("slippage", newSlippage, _slippage);
        _slippage = newSlippage;
    }
```

4.1 *changeTreasuryAddress*(**newTreasuryAddress**) X onlyGovernor

```
    function changeTreasuryAddress(address newTreasuryAddress) external onlyGovernor {
        require(newTreasuryAddress != address(0), "Treasury can't be set to 0");
        emit TreasuryChange(newTreasuryAddress, _treasuryAddress);
        _treasuryAddress = newTreasuryAddress;
    }
```

4.1 ***changeOriginationFeeRate***(**newFeeRate**) X onlyGovernor

```
    function changeOriginationFeeRate(uint256 newFeeRate) external onlyGovernor {
        // require(newFeeRate > (1 ether / 1000) && newFeeRate < (50 ether / 1000), "
            ↪ newFeeRate out of range");
        emit ParameterChange("originationFeeRate", newFeeRate, _originationFeeRate);
        _originationFeeRate = newFeeRate;
    }
```

4.1 *changeGlobalCollateralRate*(***newGlobalCollateralRate***) X onlyGovernor

```
    function changeGlobalCollateralRate(uint256 newGlobalCollateralRate) external
        ↪ onlyGovernor {
        require(newGlobalCollateralRate <= 100 && newGlobalCollateralRate != 0, "New
            ↪ collateral rate out of range");
        emit ParameterChange("globalCollateralRate", newGlobalCollateralRate,
            ↪ _globalCollateralRate);
        _globalCollateralRate = newGlobalCollateralRate;
    }
```

4.1 ***changeMaxNumberOfCycles***(*newMaxNumberOfCycles*) X onlyGovernor

```
    function changeMaxNumberOfCycles(uint256 newMaxNumberOfCycles) external onlyGovernor {
        require(newMaxNumberOfCycles < 20 && newMaxNumberOfCycles != 0, "New max n of cycles
            ↪ out of range");
        emit ParameterChange("maxNumberOfCycles", newMaxNumberOfCycles, _maxNumberOfCycles);
        _maxNumberOfCycles = newMaxNumberOfCycles;
    }
```

4.1 *changeRebaseFeeRate*(*newRebaseFeeRate*) X onlyGovernor

```
    function changeRebaseFeeRate(uint256 newRebaseFeeRate) external onlyGovernor {
        // rebaseFeeRate must be a number between 1 and 99 (in 18 decimal)
        require(newRebaseFeeRate < (100 ether) && newRebaseFeeRate > (0 ether), "New rebase
            ↪ fee rate out of range");
        emit ParameterChange("rebaseFeeRate", newRebaseFeeRate, _rebaseFeeRate);
        _rebaseFeeRate = newRebaseFeeRate;
    }
```

4.1 changeCurveMaxExchangeGuard(**newCurveMaxExchangeGuard**) X onlyGovernor

```
    function changeCurveMaxExchangeGuard(uint256 newCurveMaxExchangeGuard) external
        ↪ onlyGovernor {
        require(newCurveMaxExchangeGuard < 100 && newCurveMaxExchangeGuard > 1, "
            ↪ newCurveMaxExGuard out of range");
        emit ParameterChange("curveMaxExchangeGuard", newCurveMaxExchangeGuard,
            ↪ _curveMaxExchangeGuard);
        _curveMaxExchangeGuard = newCurveMaxExchangeGuard;
    }
```

4.1 *changeMinPositionCollateral*(***newMinPositionCollateral***) X onlyGovernor

```
    function changeMinPositionCollateral(uint256 newMinPositionCollateral) external
        ↪ onlyGovernor {
        require(newMinPositionCollateral < (1000000 ether) && newMinPositionCollateral > (1
            ↪ ether), "New min collateral out of range");
        emit ParameterChange("minPositionCollateral", newMinPositionCollateral,
            ↪ _minPositionCollateral);
        _minPositionCollateral = newMinPositionCollateral;
    }
```

4.1 changePositionTimeToLiveInDays(newPositionTimeToLiveInDays) X onlyGovernor

```
    function changePositionTimeToLiveInDays(uint256 newPositionTimeToLiveInDays) external
        ↪ onlyGovernor {
        require(newPositionTimeToLiveInDays < 10000 && newPositionTimeToLiveInDays > 30, "
            ↪ newPositionTimeToLiveInDays OOR");
        emit ParameterChange("newPositionTimeToLiveInDays", newPositionTimeToLiveInDays,
            ↪ _positionTimeToLiveInDays);
        _positionTimeToLiveInDays = newPositionTimeToLiveInDays;
    }
```

4.1 getCoordinatorLeverageBalance()

```
    function getCoordinatorLeverageBalance() external view returns (uint256) {
        return _coordinatorLeverageBalance;
    }
```

4.1 getMaxNumberOfCycles()

```
    function getMaxNumberOfCycles() external view returns (uint256) {
        return _maxNumberOfCycles;
    }
```

4.1 getOriginationFeeRate()

```
    function getOriginationFeeRate() external view returns (uint256) {
        return _originationFeeRate;
    }
```

4.1 getGlobalCollateralRate()

```
    function getGlobalCollateralRate() external view returns (uint256) {
        return _globalCollateralRate;
    }
```

4.1 getRebaseFeeRate()

```
    function getRebaseFeeRate() external view returns (uint256) {
        return _rebaseFeeRate;
    }
```

4.1 getCurveMaxExchangeGuard()

```
    function getCurveMaxExchangeGuard() external view returns (uint256) {
        return _curveMaxExchangeGuard;
    }
```

4.1 *getTreasuryAddress*()

```
    function getTreasuryAddress() external view returns (address) {
        require(_treasuryAddress != address(0), "Treasury address is not set");
        return _treasuryAddress;
    }
```

4.1 getCurveGuardPercentage()

```
    function getCurveGuardPercentage() external view returns (uint256) {
        return _curveGuardPercentage;
    }
```

4.1 *getSlippage*()

```
    function getSlippage() external view returns (uint256) {
        return _slippage;
    }
```

4.1 **getArchToLevRatio**()

```
    function getArchToLevRatio() public view returns (uint256) {
        return _auction.getCurrentBiddingPrice();
    }
```

4.1 ***getMinPositionCollateral***()

```
    function getMinPositionCollateral() external view returns (uint256) {
        return _minPositionCollateral;
    }
```

4.1 **getPositionTimeToLiveInDays**()

```
    function getPositionTimeToLiveInDays() external view returns (uint256) {
        return _positionTimeToLiveInDays;
    }
```

4.1 getAllowedLeverageForPosition(principle, *numberOfCycles*)

```
    /// Method returns the allowed pge for principle and number of cycles
    /// Return value does not include principle!
    /// must be public as we need to access it in contract
    function getAllowedLeverageForPosition(uint256 principle, uint256 numberOfCycles) public
        ↪ view returns (uint256) {
        require(numberOfCycles <= _maxNumberOfCycles, "Cycles greater than max allowed");
        uint256 leverageAmount = 0;
        uint256 cyclePrinciple = principle;
        // console.log("getAllowedLeverageForPosition principle %s, numberOfCycles %s",
            ↪ principle / 1 ether, numberOfCycles);
        for (uint256 i = 0; i < numberOfCycles; ++i) {
            // console.log("getAllowedLeverageForPosition looping on cycles");
            cyclePrinciple = (cyclePrinciple * _globalCollateralRate) / 100;
            leverageAmount += cyclePrinciple;
        }
        // console.log("getAllowedLeverageForPosition: leverageAmount %s", leverageAmount / 1
            ↪  ether);
        return leverageAmount;
    }
```

4.1 getAllowedLeverageForPositionWithArch(principle, *numberOfCycles*, archAmount)

```
    function getAllowedLeverageForPositionWithArch(
        uint256 principle,
        uint256 numberOfCycles,
        uint256 archAmount
    ) external view returns (uint256) {
        uint256 allowedLeverageNoArchLimit = getAllowedLeverageForPosition(principle,
            ↪ numberOfCycles);
        uint256 allowedLeverageWithGivenArch = calculateLeverageAllowedForArch(archAmount);
        if (allowedLeverageWithGivenArch / 10000 >= allowedLeverageNoArchLimit / 10000) {
            // In this case, user approved more(or exactly) arch tokens needed for leverage
            return allowedLeverageNoArchLimit;
        } else {
            revert("Not enough Arch for Pos");
        }
    }
```

4.1 calculateOriginationFee(leverageAmount)

```
    function calculateOriginationFee(uint256 leverageAmount) external view returns (uint256)
        ↪ {
        return (_originationFeeRate * leverageAmount) / 1 ether;
    }
```

4.1 *calculateArchNeededForLeverage*(leverageAmount)

```
    function calculateArchNeededForLeverage(uint256 leverageAmount) external view returns (
        ↪ uint256) {
        /// This method add a bit more Arch then is needed to get around integer rounding
        uint256 naturalNumberRatio = getArchToLevRatio() / 1 ether;
        return (leverageAmount / naturalNumberRatio) + 1000;
    }
```

4.1 calculateLeverageAllowedForArch(archAmount)

```
    function calculateLeverageAllowedForArch(uint256 archAmount) public view returns (uint256
        ↪ ) {
        return (getArchToLevRatio() / 1 ether) * archAmount;
    }
```

4.1 _authorizeUpgrade(newImplementation)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

4.1 *fallback*() X

```
    fallback() external {
        revert("ParamStore : Invalid access");
    }
```

Chapter 5

PoolManager

5.1 contract PoolManager

```
contract PoolManager is AccessController, ReentrancyGuardUpgradeable, UUPSUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    address internal _addressParameterStore;
    address internal _addressCoordinator;
    address internal _addressPoolLvUSD3CRV;
    IERC20Upgradeable internal _lvusd;
    IERC20Upgradeable internal _crv3;
    ParameterStore internal _paramStore;
    ICurveFiCurve internal _poolLvUSD3CRV;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

5.1 initialize() X initializer

```
    function initialize() public initializer {
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
    }
```

5.1 **setDependencies**(*addressParameterStore*, *addressCoordinator*, **addressLvUSD**, address3CRV, *addressPoolLvUSD3CRV*) X *nonReentrant* ***onlyAdmin***

```
    /**
     * @dev initialize Pool Manager
     * @param addressParameterStore ParameterStore address
     * @param addressCoordinator Coordinator contract address
     * @param addressLvUSD lvUSD ERC20 contract address
     * @param address3CRV 3CRV ERC20 contract address
     * @param addressPoolLvUSD3CRV 3CRV+LvUSD pool address
     */
    function setDependencies(
        address addressParameterStore,
        address addressCoordinator,
        address addressLvUSD,
        address address3CRV,
        address addressPoolLvUSD3CRV
    ) external nonReentrant onlyAdmin {
        // Set variables
        _addressParameterStore = addressParameterStore;
        _addressCoordinator = addressCoordinator;
        _addressPoolLvUSD3CRV = addressPoolLvUSD3CRV;

        // Load contracts
        _paramStore = ParameterStore(addressParameterStore);
        _lvusd = IERC20Upgradeable(addressLvUSD);
        _crv3 = IERC20Upgradeable(address3CRV);
        _poolLvUSD3CRV = ICurveFiCurve(addressPoolLvUSD3CRV);

        _lvusd.safeApprove(_addressPoolLvUSD3CRV, 0);
        _crv3.safeApprove(_addressPoolLvUSD3CRV, 0);

        _lvusd.safeApprove(_addressPoolLvUSD3CRV, type(uint256).max);
        _crv3.safeApprove(_addressPoolLvUSD3CRV, type(uint256).max);
    }
```

5.1 ***fundPoolWith3CRV***(*buyerAddress*, amoutToFundInLvUSD) X *nonReentrant* ***onlyAdmin***

```
    function fundPoolWith3CRV(address buyerAddress, uint256 amoutToFundInLvUSD) external
        ↪ nonReentrant onlyAdmin returns (uint256) {
        /// Method assumes that this contract , has allowance to spend buyerAddress 3CRV
            ↪ tokens
        /// Method assumes that this contract, has allowance to spend Coordinator lvUSD
            ↪ tokens
        require(_lvusd.balanceOf(_addressCoordinator) > amoutToFundInLvUSD, "Insufficient
            ↪ lvUSD on Coord");
        // // Transfer lvUSD and 3CRV to this contract
        _lvusd.safeTransferFrom(_addressCoordinator, address(this), amoutToFundInLvUSD);
        _crv3.safeTransferFrom(buyerAddress, address(this), amoutToFundInLvUSD);
        uint256[2] memory amounts = [amoutToFundInLvUSD, amoutToFundInLvUSD];
        uint256 expectedTokenAmountToGet = (_poolLvUSD3CRV.calc_token_amount(amounts, true) *
            ↪  99) / 100;
        return _poolLvUSD3CRV.add_liquidity(amounts, expectedTokenAmountToGet, buyerAddress);
    }
```

5.1 _authorizeUpgrade(newImplementation)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

Chapter 6

PositionToken

6.1 contract *PositionToken*

```
contract PositionToken is
    AccessController,
    ReentrancyGuardUpgradeable,
    ERC721Upgradeable,
    ERC721EnumerableUpgradeable,
    ERC721BurnableUpgradeable,
    UUPSUpgradeable
{
    using CountersUpgradeable for CountersUpgradeable.Counter;

    CountersUpgradeable.Counter private _positionTokenIdCounter;

    /// mapping of address to which TokenID it owns (only used for viewing methods)
    mapping(address => uint256[]) internal _addressToTokensOwnedMapping;


    event NFTCreated(uint256 indexed _positionId, address indexed _minter);
    event NFTBurned(uint256 indexed _positionId, address indexed _redeemer);

     /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

6.1 safeMint(to) X *onlyExecutive*

```
    /* Privileged functions: Executive */
    function safeMint(address to) external onlyExecutive returns (uint256 positionTokenId) {
        positionTokenId = _positionTokenIdCounter.current();
        _positionTokenIdCounter.increment();
        _safeMint(to, positionTokenId);
        _setApprovalForAll(to, getAddressExecutive(), true);
        emit NFTCreated(positionTokenId, to);
        return positionTokenId;
    }
```

6.1 **exists**(*positionTokenId*)

```
    function exists(uint256 positionTokenId) external view returns (bool) {
        return _exists(positionTokenId);
    }
```

6.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

6.1 ***initialize***() X initializer

```
    function initialize() public initializer {
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        __ERC721_init("ArchimedesPositionToken", "APNT");
        __ERC721Enumerable_init();
        __ERC721Burnable_init();
        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
    }
```

6.1 ***burn***(*positionTokenId*) X *nonReentrant* *onlyExecutive*

```
    /* override burn to only allow executive to burn positionToken */
    function burn(uint256 positionTokenId) public override(ERC721BurnableUpgradeable)
        ↪ nonReentrant onlyExecutive {
        super.burn(positionTokenId);
        emit NFTBurned(positionTokenId, msg.sender);
    }
```

6.1 **supportsInterface**(*interfaceId*)

```
    /* Override required by Solidity: */
    function supportsInterface(bytes4 interfaceId)
        public
        view
        override(ERC721Upgradeable, ERC721EnumerableUpgradeable, AccessControlUpgradeable)
        returns (bool)
    {
        return super.supportsInterface(interfaceId);
    }
```

6.1 _beforeTokenTransfer(from, to, tokenId)

```
    /* Override required by Solidity: */
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual override(ERC721Upgradeable, ERC721EnumerableUpgradeable) {
        uint256 tokenIdArrayIndex;
        // remove prev owner from _addressToTokensOwnedMapping only if this is not from this
            ↪ contract (ie new position) and mapping exist
        // console.log("_beforeTokenTransfer from %s, to %s, tokenId %s", from, to, tokenId);
        if (from != address(this) && _addressToTokensOwnedMapping[from].length != 0) {
            // console.log("_beforeTokenTransfer from %s, to %s, tokenId %s", from, to,
                ↪ tokenId);
            for (uint256 i = 0; i < _addressToTokensOwnedMapping[from].length; i++) {
                if (_addressToTokensOwnedMapping[from][i] == tokenId) {
                    tokenIdArrayIndex = i;
                    uint256 lastTokenIdInArray = _addressToTokensOwnedMapping[from][
                        ↪ _addressToTokensOwnedMapping[from].length - 1];
                    _addressToTokensOwnedMapping[from][tokenIdArrayIndex] =
                        ↪ lastTokenIdInArray;
                    _addressToTokensOwnedMapping[from].pop();
                    break;
                }
            }
        }

        return super._beforeTokenTransfer(from, to, tokenId);
    }
```

6.1 _afterTokenTransfer(from, to, tokenId)

```
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual override(ERC721Upgradeable) {
        super._afterTokenTransfer(from, to, tokenId);

        // Add tokenID from To address to _addressToTokensOwnedMapping
        // console.log("AfterTokenTransfer from %s, to %s, tokenId %s", from, to, tokenId);
        if (to != address(0)) {
            _addressToTokensOwnedMapping[to].push(tokenId);
        }

        /// set approval for executive to interact with tokenID
        _setApprovalForAll(to, getAddressExecutive(), true);
    }
```

6.1 *getTokenIDsArray*(owner)

```
function getTokenIDsArray(address owner) external view returns (uint256[] memory) {
    uint256[] memory arrayToReturn = _addressToTokensOwnedMapping[owner];
    return arrayToReturn;
}
```

6.1 _authorizeUpgrade(newImplementation)

```
// solhint-disable-next-line
function _authorizeUpgrade(address newImplementation) internal override {
    _requireAdmin();
}
```

6.1 *fallback*() X

```
fallback() external {
    revert("PositionToken : Invalid access");
}
```

Chapter 7

Exchanger

7.1 contract Exchanger

```
/// TODO Approval & Allownace should NOT BE MAX VALUES for pools
/// Use the overloaded function with TO parameter for exchange

/// @title Exchanger
/// @dev is in charge of interacting with the CurveFi pools
contract Exchanger is AccessController, ReentrancyGuardUpgradeable, IExchanger,
    ↪ UUPSUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    address internal _addressParameterStore;
    address internal _addressCoordinator;
    address internal _addressPoolLvUSD3CRV;
    address internal _addressPool0USD3CRV;
    IERC20Upgradeable internal _lvUSD;
    IERC20Upgradeable internal _ousd;
    IERC20Upgradeable internal _crv3;
    ICurveFiCurve internal _poolLvUSD3CRV;
    ICurveFiCurve internal _pool0USD3CRV;

    ParameterStore internal _paramStore;
    int128 internal _indexLvUSD;
    int128 internal _indexOUSD;
    int128 internal _index3CRV;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;

    // /** @dev curve stable metapools provide 1:1 swaps
    //  * if the pools are very bent, this is a protection for users
    //  * TODO: user should be able to override and force a trade
    //  * @dev expressed as a percentage
    //  * 100 would require a perfect 1:1 swap
    //  * 90 allows at most, 1:.9 swaps
    //  */
    // uint256 internal _curveGuardPercentage;
}
```

7.1 **setDependencies**(*addressParameterStore*, *addressCoordinator*, *addressLvUSD*, *addressOUSD*, address3CRV, *addressPoolLvUSD3CRV*, *addressPoolOUSD3CRV*) X *nonReentrant* *onlyAdmin*

```
/**
 * @dev initialize Exchanger
 * @param addressParameterStore ParameterStore address
 * @param addressCoordinator Coordinator contract address
 * @param addressLvUSD lvUSD ERC20 contract address
 * @param addressOUSD OUSD ERC20 contract address
 * @param address3CRV 3CRV ERC20 contract address
 * @param addressPoolLvUSD3CRV 3CRV+LvUSD pool address
 * @param addressPoolOUSD3CRV 3CRV+OUSD pool address
 */
function setDependencies(
    address addressParameterStore,
    address addressCoordinator,
    address addressLvUSD,
    address addressOUSD,
    address address3CRV,
    address addressPoolLvUSD3CRV,
    address addressPoolOUSD3CRV
) external nonReentrant onlyAdmin {
    require(addressParameterStore != address(0), "cant set to 0 A");
    require(addressCoordinator != address(0), "cant set to 0 A");
    require(addressLvUSD != address(0), "cant set to 0 A");
    require(addressOUSD != address(0), "cant set to 0 A");
    require(address3CRV != address(0), "cant set to 0 A");
    require(addressPoolLvUSD3CRV != address(0), "cant set to 0 A");
    require(addressPoolOUSD3CRV != address(0), "cant set to 0 A");

    // Set variables
    _addressParameterStore = addressParameterStore;
    _addressCoordinator = addressCoordinator;
    _addressPoolLvUSD3CRV = addressPoolLvUSD3CRV;
    _addressPoolOUSD3CRV = addressPoolOUSD3CRV;

    // Load contracts
    _paramStore = ParameterStore(addressParameterStore);
    _lvUSD = IERC20Upgradeable(addressLvUSD);
    _ousd = IERC20Upgradeable(addressOUSD);
    _crv3 = IERC20Upgradeable(address3CRV);
    _poolLvUSD3CRV = ICurveFiCurve(addressPoolLvUSD3CRV);
    _poolOUSD3CRV = ICurveFiCurve(addressPoolOUSD3CRV);
}
```

7.1 constructor() X

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

7.1 *initialize*() X initializer

```
function initialize() public initializer {
    __AccessControl_init();
    __ReentrancyGuard_init();
    __UUPSUpgradeable_init();

    _grantRole(ADMIN_ROLE, _msgSender());
    setGovernor(_msgSender());
    setExecutive(_msgSender());
    setGuardian(_msgSender());

    _indexLvUSD = 0;
    _indexOUSD = 0;
    _index3CRV = 1;
}
```

7.1 _exchangerLvUSDBurnOnUnwind(amount)

```
function _exchangerLvUSDBurnOnUnwind(uint256 amount) internal {
    /// Is it possible to exploit via transferring lvUSD to exchanger which then go back
        ↪ to coordinator?
    uint256 currentExchangerLvUSDBalance = _lvUSD.balanceOf(address(this));
    require(currentExchangerLvUSDBalance >= amount, "insuf lvUSD to trnsf to Exhanger");
    ERC20Burnable(address(_lvUSD)).burn(amount);
}
```

7.1 *swapOUSDforLvUSD*(amountOUSD, *minRequiredLvUSD*) X *nonReentrant* *onlyExecutive*

```
/**
 * @dev Exchanges OUSD for LvUSD using multiple CRV3Metapools
 * returns amount of LvUSD
 * - MUST emit an event
 * - MUST revert if we dont get back the minimum required OUSD
 * @param amountOUSD amount of OUSD we have available to exchange
 * @param minRequiredLvUSD amount of OUSD we must get back or revert
 * @return lvUSDReturned amount of LvUSD we got back
 * NOTE: lvUSDReturned isnt necessarily minRequiredLvUSD - it
 * will be at least that much based on pool price variations
 * @return remainingOUSD amount of left over OUSD after the exchange
 * NOTE: There is no gaurnatee of a 1:1 exchange ratio
 * @dev OUSD funds are already under Exchanger address, if called by Coordinator
 */
function swapOUSDforLvUSD(uint256 amountOUSD, uint256 minRequiredLvUSD)
    external
    nonReentrant
    onlyExecutive
    returns (uint256 lvUSDReturned, uint256 remainingOUSD)
{
    return _swapOUSDforLvUSD(amountOUSD, minRequiredLvUSD);
}
```

7.1 swapLvUSDforOUSD(amountLvUSD) X nonReentrant onlyExecutive

```
/**
 * @dev Exchanges LvUSD for OUSD using multiple CRV3Metapools
 * @param amountLvUSD amount of LvUSD we will put in
 * @return amountOUSD amount of OUSD returned from exchange
 * - MUST emit an event
 * NOTE: There is no guarantee of a 1:1 exchange ratio, but should be close
 * Minimum is 90% * 90%  / _curveGuardPercentage * _curveGuardPercentage
 */
function swapLvUSDforOUSD(uint256 amountLvUSD) external nonReentrant onlyExecutive
    ↪ returns (uint256 amountOUSD) {
    return _swapLvUSDforOUSD(amountLvUSD);
}
```

7.1 _swapOUSDforLvUSD(amountOUSD, minRequiredLvUSD)

```
// Send OUSD, get lvUSD back and the reminder of OUSD
function _swapOUSDforLvUSD(uint256 amountOUSD, uint256 minRequiredLvUSD) internal returns
    ↪  (uint256 lvUSDReturned, uint256 remainingOUSD) {
    /// process is go to OUSD/3CRV pool, exchange as much OUSD as needed for enough 3CRV.
        ↪  Exhange all the 3CRV you got for lvUSD on lvUSD/3CRV pool
    // Get the amount of 3CRV gotten from exhanging minRequiredLvUSD or lvUSD to 3CRV.
        ↪ This is actually the other way around then what we will actually do. Used as
        ↪ an indicator
    uint256 _needed3CRV = _poolLvUSD3CRV.get_dy(0, 1, minRequiredLvUSD);
    // Get the amount of OUSD gotten from exhanging above amount of 3CRV on OUSD/3CRV
        ↪ pool
    uint256 _neededOUSD = _poolOUSD3CRV.get_dy(1, 0, _needed3CRV);
    // Add small buffer to needed OUSD and calculate in the right order (ie first exhange
        ↪  OUSD for 3CRV, then exhange that 3CRV for lvUSD)
    /// Notice that the small slippage is static here. Further below when we actaully
        ↪ exhange funds we use the user defined slippage.
    _neededOUSD = (_neededOUSD * 1005) / 1000; // This will fix lower balances slippages
    uint256 _obtained3CRV = _poolOUSD3CRV.get_dy(0, 1, _neededOUSD);
    uint256 _obtainedLvUSD = _poolLvUSD3CRV.get_dy(1, 0, _obtained3CRV);
    /// if the amount of expected lvUSD (_obtainedLvUSD) is lower then the min amount of
        ↪ lvUSD we expect to get back, re-calculate
    // the important output of this code block is the correct amount of _neededOUSD to
        ↪ exhange through the flow of the two pools.
    if (_obtainedLvUSD < minRequiredLvUSD) {
        // _difference will give us the delta of lvUSD we need to get (which means using
            ↪ more OUSD)
        uint256 _difference = (minRequiredLvUSD) - _obtainedLvUSD + 10**18; // +1 just in
            ↪  case
        uint256 _crv3Difference = _poolOUSD3CRV.get_dy(0, 1, _difference);
        uint256 _lvUSDDifference = _poolLvUSD3CRV.get_dy(1, 0, _crv3Difference);

        uint256 finalAmount = _obtainedLvUSD + _lvUSDDifference;
        _neededOUSD = _neededOUSD + _difference;

        /// Do same correction cycle as above again.
        if (finalAmount < (minRequiredLvUSD)) {
            // console.log("Inside calc finalAmount");
            _difference = (minRequiredLvUSD) - finalAmount + 10**18; // +1 just in case
            _crv3Difference = _poolOUSD3CRV.get_dy(0, 1, _difference);
            _lvUSDDifference = _poolLvUSD3CRV.get_dy(1, 0, _crv3Difference);

            finalAmount = finalAmount + _lvUSDDifference;
            _neededOUSD = _neededOUSD + _difference;
        }
        // console.log("_swapOUSDforLvUSD_inside if: _neededOUSD %s, finalAmount(ofLUSD)
            ↪ %s", _neededOUSD / 1 ether, finalAmount / 1 ether);
    }
```

```
        // console.log("_swapOUSDforLvUSD1 : _neededOUSD %s, _obtainedLvUSD %s", _neededOUSD
            ↪ / 1 ether, _obtainedLvUSD / 1 ether);
        require(amountOUSD >= _neededOUSD, "Not enough OUSD for exchange");

        // We lose some $ from fees and slippage
        // multiply _neededOUSD * 103%
        uint256 _returned3CRV = _xOUSDfor3CRV(_neededOUSD);

        uint256 _returnedLvUSD = _x3CRVforLvUSD(_returned3CRV);
        require(_returnedLvUSD >= minRequiredLvUSD, "3/lv insuf eX to lvUSD");

        // calculate remaining OUSD
        remainingOUSD = amountOUSD - _neededOUSD;
        _ousd.safeTransfer(_addressCoordinator, remainingOUSD);

        // send all swapped lvUSD to coordinator
        _exchangerLvUSDBurnOnUnwind(_returnedLvUSD);

        return (_returnedLvUSD, remainingOUSD);
    }
```

7.1 _swapLvUSDforOUSD(amountLvUSD)

```
    function _swapLvUSDforOUSD(uint256 amountLvUSD) internal returns (uint256 amountOUSD) {
        uint256 _returned3CRV = _xLvUSDfor3CRV(amountLvUSD);
        uint256 _returnedOUSD = _x3CRVforOUSD(_returned3CRV);
        _ousd.safeTransfer(_addressCoordinator, _returnedOUSD);
        return _returnedOUSD;
    }
```

7.1 _xLvUSDfor3CRV(amountLvUSD)

```
    /**
     * @dev Exchange using the CurveFi LvUSD/3CRV Metapool
     * @param amountLvUSD amount of LvUSD to exchange
     * @return amount3CRV amount of 3CRV returned from exchange
     */
    function _xLvUSDfor3CRV(uint256 amountLvUSD) internal returns (uint256 amount3CRV) {
        /**
         * _expected3CRV uses get_dy() to estimate amount the exchange will give us
         * _minimum3CRV minimum accounting for slippage. (_expected3CRV * slippage)
         * _returned3CRV amount we actually get from the pool
         * _guard3CRV sanity check to protect user
         */
        uint256 _expected3CRV;
        uint256 _minimum3CRV;
        uint256 _returned3CRV;
        uint256 _guard3CRV = (amountLvUSD * _paramStore.getCurveGuardPercentage()) / 100;

        // Verify Exchanger has enough LvUSD to use
        require(amountLvUSD <= _lvUSD.balanceOf(address(this)), "Insufficient LvUSD in
            ↪ Exchanger.");

        // Estimate expected amount of 3CRV
        // get_dy(indexCoinSend, indexCoinRec, amount)
        _expected3CRV = _poolLvUSD3CRV.get_dy(0, 1, amountLvUSD);

        // /// Make sure expected3CRV is not too high!
        _checkExchangeExpectedReturnInLimit(amountLvUSD, _expected3CRV);

        // Set minimum required accounting for slippage
        _minimum3CRV = (_expected3CRV * (100 - _paramStore.getSlippage())) / 100;
```

```
        // Make sure pool isn't too bent
        // TODO allow user to override this protection
        // TODO auto balance if pool is bent
        // console.log("Exchanger:req  _minimum3CRV >= _guard3CRV, %s >= %s", _minimum3CRV,
            ↪ _guard3CRV);
        require(_minimum3CRV >= _guard3CRV, "LvUSD pool too imbalanced.");

        // Increase allowance
        _lvUSD.safeIncreaseAllowance(address(_poolLvUSD3CRV), amountLvUSD);

        // Exchange LvUSD for 3CRV:
        _returned3CRV = _poolLvUSD3CRV.exchange(0, 1, amountLvUSD, _minimum3CRV);

        // Set approval to zero for safety
        _lvUSD.safeApprove(address(_poolLvUSD3CRV), 0);

        return _returned3CRV;
    }
```

7.1 _xOUSDfor3CRV(amountOUSD)

```
    /**
     * @dev Exchange using the CurveFi OUSD/3CRV Metapool
     * @param amountOUSD amount of OUSD to put into the pool
     * @return amount3CRV amount of 3CRV returned from exchange
     */
    function _xOUSDfor3CRV(uint256 amountOUSD) internal returns (uint256 amount3CRV) {
        /**
         * @param _expected3CRV uses get_dy() to estimate amount the exchange will give us
         * @param _minimum3CRV minimum accounting for slippage. (_expected3CRV * slippage)
         * @param _returned3CRV amount we actually get from the pool
         * @param _guard3CRV sanity check to protect user
         */
        uint256 _expected3CRV;
        uint256 _minimum3CRV;
        uint256 _returned3CRV;
        uint256 _guard3CRV = (amountOUSD * _paramStore.getCurveGuardPercentage()) / 100;

        // Verify Exchanger has enough OUSD to use
        // console.log("amountOUSD <= _ousd.balanceOf(address(this) %s <= %s", amountOUSD,
            ↪ _ousd.balanceOf(address(this)));
        require(amountOUSD <= _ousd.balanceOf(address(this)), "Insufficient OUSD in Exchanger
            ↪ .");

        // Estimate expected amount of 3CRV
        // get_dy(indexCoinSend, indexCoinRec, amount)
        _expected3CRV = _poolOUSD3CRV.get_dy(0, 1, amountOUSD);

        // Set minimum required accounting for slippage
        _minimum3CRV = (_expected3CRV * (100 - _paramStore.getSlippage())) / 100;

        // Make sure pool isn't too bent
        // TODO allow user to override this protection
        // TODO auto balance if pool is bent
        require(_minimum3CRV >= _guard3CRV, "OUSD pool too imbalanced.");

        // Increase allowance
        _ousd.safeIncreaseAllowance(address(_poolOUSD3CRV), amountOUSD);

        // Exchange OUSD for 3CRV:
        _returned3CRV = _poolOUSD3CRV.exchange(0, 1, amountOUSD, _minimum3CRV);

        // Set approval to zero for safety
        _ousd.safeApprove(address(_poolOUSD3CRV), 0);
```

```
        return _returned3CRV;
    }
```

7.1 _x3CRVforLvUSD(amount3CRV)

```
    /**
     * @dev Exchange using the CurveFi LvUSD/3CRV Metapool
     * @param amount3CRV amount of 3CRV to exchange
     * @return amountLvUSD amount of LvUSD returned from exchange
     */
    function _x3CRVforLvUSD(uint256 amount3CRV) internal returns (uint256 amountLvUSD) {
        /**
         * @param _expectedLvUSD uses get_dy() to estimate amount the exchange will give us
         * @param _minimumLvUSD minimum accounting for slippage. (_expectedLvUSD * slippage)
         * @param _returnedLvUSD amount we actually get from the pool
         * @param _guardLvUSD sanity check to protect user
         */
        uint256 _expectedLvUSD;
        uint256 _minimumLvUSD;
        uint256 _returnedLvUSD;
        uint256 _guardLvUSD = (amount3CRV * _paramStore.getCurveGuardPercentage()) / 100;

        // Verify Exchanger has enough 3CRV to use
        require(amount3CRV <= _crv3.balanceOf(address(this)), "Insufficient 3CRV in Exchanger
            ↪ .");

        // Estimate expected amount of 3CRV
        // get_dy(indexCoinSend, indexCoinRec, amount)
        _expectedLvUSD = _poolLvUSD3CRV.get_dy(1, 0, amount3CRV);

        // Set minimum required accounting for slippage
        _minimumLvUSD = (_expectedLvUSD * (100 - _paramStore.getSlippage())) / 100;

        // Make sure pool isn't too bent
        // TODO allow user to override this protection
        // TODO auto balance if pool is bent
        require(_minimumLvUSD >= _guardLvUSD, "LvUSD pool too imbalanced.");

        // Increase allowance
        _crv3.safeIncreaseAllowance(address(_poolLvUSD3CRV), amount3CRV);

        // Exchange 3CRV for LvUSD:
        _returnedLvUSD = _poolLvUSD3CRV.exchange(1, 0, amount3CRV, _minimumLvUSD);

        // Set approval to zero for safety
        _crv3.safeApprove(address(_poolLvUSD3CRV), 0);

        return _returnedLvUSD;
    }
```

7.1 _x3CRVforOUSD(amount3CRV)

```
    /**
     * @dev Exchange using the CurveFi OUSD/3CRV Metapool
     * @param amount3CRV amount of LvUSD to exchange
     * @return amountOUSD amount returned from exchange
     */
    function _x3CRVforOUSD(uint256 amount3CRV) internal returns (uint256 amountOUSD) {
        /**
         * @param _expectedOUSD uses get_dy() to estimate amount the exchange will give us
         * @param _minimumOUSD minimum accounting for slippage. (_expectedOUSD * slippage)
         * @param _returnedOUSD amount we actually get from the pool
         * @param _guardOUSD sanity check to protect user
         */
        uint256 _expectedOUSD;
        uint256 _minimumOUSD;
        uint256 _returnedOUSD;
        uint256 _guardOUSD = (amount3CRV * _paramStore.getCurveGuardPercentage()) / 100;

        // Verify Exchanger has enough 3CRV to use
        require(amount3CRV <= _crv3.balanceOf(address(this)), "Insufficient 3CRV in Exchanger
            ↪ .");

        // Estimate expected amount of 3CRV
        // get_dy(indexCoinSend, indexCoinRec, amount)
        _expectedOUSD = _poolOUSD3CRV.get_dy(1, 0, amount3CRV);

        // Set minimum required accounting for slippage
        _minimumOUSD = (_expectedOUSD * (100 - _paramStore.getSlippage())) / 100;

        // Make sure pool isn't too bent
        // TODO allow user to override this protection
        // TODO auto balance if pool is bent
        require(_minimumOUSD >= _guardOUSD, "LvUSD pool too imbalanced.");

        // Increase allowance
        _crv3.safeIncreaseAllowance(address(_poolOUSD3CRV), amount3CRV);

        // Exchange LvUSD for 3CRV:
        _returnedOUSD = _poolOUSD3CRV.exchange(1, 0, amount3CRV, _minimumOUSD);

        // Set approval to zero for safety
        _crv3.safeApprove(address(_poolOUSD3CRV), 0);

        return _returnedOUSD;
    }
```

7.1 _checkExchangeExpectedReturnInLimit(amountToExchange, expctedExchangeReturn)

```
    function _checkExchangeExpectedReturnInLimit(uint256 amountToExchange, uint256
        ↪ expctedExchangeReturn) internal {
        uint256 maxAllowedExchangeReturn = amountToExchange + (amountToExchange * _paramStore
            ↪ .getCurveMaxExchangeGuard()) / 100;
        require(expctedExchangeReturn <= maxAllowedExchangeReturn, "Expected return value too
            ↪  big");
    }
```

7.1 _authorizeUpgrade(newImplementation)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

7.1 estimateOusdReturnedOnUnwindMinusInterest(amountOUSD, minRequiredLvUSD)

```
    function estimateOusdReturnedOnUnwindMinusInterest(uint256 amountOUSD, uint256
        ↪ minRequiredLvUSD) external view returns (uint256) {
        uint256 _needed3CRV = _poolLvUSD3CRV.get_dy(0, 1, minRequiredLvUSD);
        uint256 _neededOUSD = _poolOUSD3CRV.get_dy(1, 0, _needed3CRV);
        // console.log("estimateOusdReturnedOnUnwind 1: _needed3CRV %s, _neededOUSD %s",
            ↪ _needed3CRV / 1 ether, _neededOUSD / 1 ether);

        _neededOUSD = (_neededOUSD * 1005) / 1000; // This will fix lower balances slippages
        uint256 _obtained3CRV = _poolOUSD3CRV.get_dy(0, 1, _neededOUSD);
        uint256 _obtainedLvUSD = _poolLvUSD3CRV.get_dy(1, 0, _obtained3CRV);

        if (_obtainedLvUSD < (minRequiredLvUSD)) {
            uint256 _difference = (minRequiredLvUSD) - _obtainedLvUSD + 10**18; // +1 just in
                ↪  case
            uint256 _crv3Difference = _poolOUSD3CRV.get_dy(0, 1, _difference);
            uint256 _lvUSDDifference = _poolLvUSD3CRV.get_dy(1, 0, _crv3Difference);

            uint256 finalAmount = _obtainedLvUSD + _lvUSDDifference;
            _neededOUSD = _neededOUSD + _difference;

            if (finalAmount < (minRequiredLvUSD)) {
                _difference = (minRequiredLvUSD) - finalAmount + 10**18; // +1 just in case
                _crv3Difference = _poolOUSD3CRV.get_dy(0, 1, _difference);
                _lvUSDDifference = _poolLvUSD3CRV.get_dy(1, 0, _crv3Difference);

                finalAmount = finalAmount + _lvUSDDifference;
                _neededOUSD = _neededOUSD + _difference;
            }
        }
        return amountOUSD - _neededOUSD;
    }
```

7.1 fallback() X

```
    fallback() external {
        revert("Exchanger : Invalid access");
    }
```

7.1 interface IExchanger

```
interface IExchanger {
}
```

7.1 **swapLvUSDforOUSD**(*amountLvUSD*) X [IExchanger]

```
/**
 * @dev Exchanges LvUSD for OUSD using multiple CRV3Metapools
 * returns amount of OUSD
 * - MUST emit an event
 * NOTE: There is no gaurnatee of a 1:1 exchange ratio
 */
function swapLvUSDforOUSD(uint256 amountLvUSD) external returns (uint256);
```

7.1 *swapOUSDforLvUSD*(**amountOUSD**, ***minRequired***) X [IExchanger]

```
/**
 * @dev Exchanges OUSD for LvUSD using multiple CRV3Metapools
 * returns amount of LvUSD
 * - MUST emit an event
 * - MUST revert if we dont get back the minimum required OUSD
 * NOTE: There is no gaurnatee of a 1:1 exchange ratio
 */
function swapOUSDforLvUSD(uint256 amountOUSD, uint256 minRequired) external returns (
    ↪ uint256 lvUSDReturned, uint256 remainingOUSD);
```

Chapter 8

Auction

8.1 contract Auction

```
contract Auction is IAuction, AccessController, UUPSUpgradeable {
    uint256 internal _currentAuctionId;
    uint256 internal _startBlock;
    uint256 internal _endBlock;
    uint256 internal _startPrice;
    uint256 internal _endPrice;

    bool internal _isAuctionClosed;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

8.1 startAuctionWithLength(length, startPrice, endPrice) X onlyAuctioneer

```
    function startAuctionWithLength(
        uint256 length,
        uint256 startPrice,
        uint256 endPrice
    ) external override onlyAuctioneer {
        uint256 endBlock = block.number + length;
        _startAuction(endBlock, startPrice, endPrice);
    }
```

8.1 startAuction(endBlock, startPrice, endPrice) X onlyAuctioneer

```
    function startAuction(
        uint256 endBlock,
        uint256 startPrice,
        uint256 endPrice
    ) external override onlyAuctioneer {
        _startAuction(endBlock, startPrice, endPrice);
    }
```

8.1 _startAuction(endBlock, startPrice, endPrice)

```
    function _startAuction(
        uint256 endBlock,
        uint256 startPrice,
        uint256 endPrice
    ) internal {
        require(isAuctionClosed() == true, "err:auction currently running");
        _validateAuctionParams(endBlock, startPrice, endPrice);
        _setAuctionPrivateMembers(endBlock, startPrice, endPrice);
        _emitAuctionStart();
        _isAuctionClosed = false;
    }
```

8.1 stopAuction() X onlyAuctioneer

```
    function stopAuction() external onlyAuctioneer {
        _isAuctionClosed = true;
        _emitAuctionForcedStopped();
    }
```

8.1 getCurrentBiddingPrice()

```
    function getCurrentBiddingPrice() external view override returns (uint256
        ↪ auctionBiddingPrice) {
        /// If reached endBlock , handle auction that is "closed"
        /// ELSE calculate current price for an open auction.
        // console.log("endBlock %s currentBlock %s", _endBlock, block.number);
        uint256 biddingPrice;
        if (isAuctionClosed()) {
            biddingPrice = _getCurrentPriceClosedAuction();
        } else {
            biddingPrice = _calcCurrentPriceOpenAuction();
        }
        // sanity and security check
        if (biddingPrice == 0) {
            revert("err:biddingPrice cant be 0");
        } else {
            return biddingPrice;
        }
    }
```

8.1 _getCurrentPriceClosedAuction()

```
    /// calc methods

    function _getCurrentPriceClosedAuction() internal view returns (uint256 auctionPrice) {
        return _endPrice;
    }
```

8.1 _calcCurrentPriceOpenAuction()

```
    function _calcCurrentPriceOpenAuction() internal view returns (uint256 auctionPrice) {
        /// y = ax + b
        /// => y = current auction price.
        /// linear graph show price in Y and price is going down over time so we'll need y =
            ↪ -ax + b
        /// might be easier to think about this as y = b - ax
        /// time is the X axis here so when we start auction t=0, when we end t=delt(
            ↪ startBlock, endBlock)
        /// we want to get time that is between 0 and 1 so we'll do
        /// so this means t_current = (currentBlock - startBlock)/delta(startBlock, endBlock)
        /// b = startPrice. b has to equal startPrice since t=0 at that point
        /// a = (startingPrice - endPrice)
        // currentPrice =  b - ax = startPrice - (startingPrice - endPrice) * t(0...1 only)
        uint256 deltaInPrices = _endPrice - _startPrice;
        uint256 deltaInPriceMulCurrentTime = (deltaInPrices * (block.number - _startBlock)) /
            ↪  (_endBlock - _startBlock);
        uint256 maxPriceForAuction = _startPrice;
        uint256 currentPrice = maxPriceForAuction + deltaInPriceMulCurrentTime;

        return currentPrice;
    }
```

8.1 isAuctionClosed()

```
    /// helper methods

    function isAuctionClosed() public view returns (bool) {
        if (_isAuctionClosed == true || _endBlock < block.number) {
            return true;
        } else {
            return false;
        }
    }
```

8.1 _validateAuctionParams(endBlock, startPrice, endPrice)

```
    function _validateAuctionParams(
        uint256 endBlock,
        uint256 startPrice,
        uint256 endPrice
    ) internal view {
        require(endBlock > block.number, "err:endBlock<=block.number");
        require(startPrice > 0, "err:start price cant be 0");
        require(startPrice < endPrice, "err:startPrice>endPrice");
    }
```

8.1 _setAuctionPrivateMembers(endBlock, startPrice, endPrice)

```
    function _setAuctionPrivateMembers(
        uint256 endBlock,
        uint256 startPrice,
        uint256 endPrice
    ) internal {
        _currentAuctionId = _currentAuctionId + 1;
        _startBlock = block.number;
        _endBlock = endBlock;
        _startPrice = startPrice;
        _endPrice = endPrice;
    }
```

8.1 _emitAuctionStart()

```
    function _emitAuctionStart() internal {
        emit AuctionStart(_currentAuctionId, _startBlock, _endBlock, _startPrice, _endPrice);
    }
```

8.1 _emitAuctionForcedStopped()

```
    function _emitAuctionForcedStopped() internal {
        emit AuctionForcedStopped(_currentAuctionId);
    }
```

8.1 initialize() X initializer

```
    /// Deplyment functionality
    function initialize() public initializer {
        __AccessControl_init();
        __UUPSUpgradeable_init();
        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
        setAuctioneer(_msgSender());
        _isAuctionClosed = true;
    }
```

8.1 _authorizeUpgrade(newImplementation)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

8.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

8.1 interface IAuction

```
interface IAuction {
    event AuctionStart(uint256 auctionId, uint256 startBlock, uint256 endBlock, uint256
        ↪ startPrice, uint256 endPrice);
    event AuctionForcedStopped(uint256 auctionId);
}
```

8.1 startAuctionWithLength(length, startPrice, endPrice) X [IAuction]

```
    function startAuctionWithLength(
        uint256 length,
        uint256 startPrice,
        uint256 endPrice
    ) external;
```

8.1 startAuction(endBlock, startPrice, endPrice) X [IAuction]

```
    function startAuction(
        uint256 endBlock,
        uint256 startPrice,
        uint256 endPrice
    ) external;
```

8.1 stopAuction() X [IAuction]

```
    function stopAuction() external;
```

8.1 getCurrentBiddingPrice() [IAuction]

```
    function getCurrentBiddingPrice() external view returns (uint256 auctionBiddingPrice);
```

Chapter 9

LeverageEngine

9.1 contract *LeverageEngine*

```
contract LeverageEngine is AccessController, ReentrancyGuardUpgradeable, UUPSUpgradeable,
    ↪ PausableUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    address internal _addressCoordinator;
    address internal _addressPositionToken;
    address internal _addressParameterStore;
    address internal _addressArchToken;
    address internal _addressOUSD;

    ICoordinator internal _coordinator;
    PositionToken internal _positionToken;
    ParameterStore internal _parameterStore;
    ArchToken internal _archToken;
    IERC20Upgradeable internal _ousd;

    address internal _addressCDP;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[43] private __gap;

    event PositionCreated(
        address indexed _from,
        uint256 indexed _positionId,
        uint256 _princple,
        uint256 _levTaken,
        uint256 _archBurned,
        uint256 _positionExp
    );
    event PositionUnwind(address indexed _from, uint256 indexed _positionId, uint256
        ↪ _positionWindfall);
}
```

9.1 **setDependencies**(*addressCoordinator*, ***addressPositionToken***, ***addressParameterStore***, addressArchToken, ***addressOUSD***, **addressCDP**) X *nonReentrant* ***onlyAdmin***

```
/// @dev set the addresses for Coordinator, PositionToken, ParameterStore
function setDependencies(
    address addressCoordinator,
    address addressPositionToken,
    address addressParameterStore,
    address addressArchToken,
    address addressOUSD,
    address addressCDP
) external nonReentrant onlyAdmin {
    require(addressCoordinator != address(0), "cant set to 0 A");
    require(addressPositionToken != address(0), "cant set to 0 A");
    require(addressParameterStore != address(0), "cant set to 0 A");
    require(addressArchToken != address(0), "cant set to 0 A");
    require(addressOUSD != address(0), "cant set to 0 A");
    _addressCoordinator = addressCoordinator;
    _coordinator = ICoordinator(addressCoordinator);
    _addressPositionToken = addressPositionToken;
    _positionToken = PositionToken(addressPositionToken);
    _addressParameterStore = addressParameterStore;
    _parameterStore = ParameterStore(addressParameterStore);
    _addressArchToken = addressArchToken;
    _archToken = ArchToken(addressArchToken);
    _addressOUSD = addressOUSD;
    _ousd = IERC20Upgradeable(_addressOUSD);
    _addressCDP = addressCDP;
}
```

9.1 *createLeveragedPosition*(ousdPrinciple, *cycles*, **maxArchAmount**, **minLeverageAmount**) X *nonReentrant* whenNotPaused

```
/// @dev deposit OUSD under NFT ID
///
/// User sends OUSD to the contract.
/// We mint NFT, assign to msg.sender and do the leverage cycles
///
/// @param ousdPrinciple the amount of OUSD sent to Archimedes
/// @param cycles How many leverage cycles to do
/// @param maxArchAmount max amount of Arch tokens to burn for position
function createLeveragedPosition(
    uint256 ousdPrinciple,
    uint256 cycles,
    uint256 maxArchAmount,
    uint256 minLeverageAmount
) external nonReentrant whenNotPaused returns (uint256) {
    return _createLeveragedPosition(ousdPrinciple, cycles, maxArchAmount, msg.sender,
        ↪ minLeverageAmount);
}
```

9.1 createLeveragedPositionFromZapper(ousdPrinciple, cycles, maxArchAmount, userAddress, minLeverageAmount) X nonReentrant whenNotPaused

```
function createLeveragedPositionFromZapper(
    uint256 ousdPrinciple,
    uint256 cycles,
    uint256 maxArchAmount,
    address userAddress,
    uint256 minLeverageAmount
) external nonReentrant whenNotPaused returns (uint256) {
    return _createLeveragedPosition(ousdPrinciple, cycles, maxArchAmount, userAddress,
        ↪ minLeverageAmount);
}
```

9.1 _createLeveragedPosition(ousdPrinciple, cycles, maxArchAmount, userAddress, minLeverageAmount)

```
/* Non-privileged functions */

function _createLeveragedPosition(
    uint256 ousdPrinciple,
    uint256 cycles,
    uint256 maxArchAmount,
    address userAddress,
    uint256 minLeverageAmount
) internal returns (uint256) {
    if (cycles == 0 || cycles > _parameterStore.getMaxNumberOfCycles()) {
        revert("Invalid number of cycles");
    }
    if (ousdPrinciple < _parameterStore.getMinPositionCollateral()) {
        revert("Collateral lower then min");
    }
    // this is how much lvUSD we can get with the given (max) arch
    uint256 lvUSDAmount = _parameterStore.getAllowedLeverageForPositionWithArch(
        ↪ ousdPrinciple, cycles, maxArchAmount);
    /// this is how much lvUSD we can get if we had "more then enough" Arch token to open
        ↪  a big position
    uint256 lvUSDAmountNeedForArguments = _parameterStore.getAllowedLeverageForPosition(
        ↪ ousdPrinciple, cycles);

    /// check that user gave enough arch allowance for cycle-principle combo
    require(lvUSDAmountNeedForArguments - 1 <= lvUSDAmount, "cant get enough lvUSD");
    uint256 archNeededToBurn = (_parameterStore.calculateArchNeededForLeverage(
        ↪ lvUSDAmount) / 10000) * 10000; //  max minus 1000 wei

    require(archNeededToBurn <= maxArchAmount, "Not enough Arch given for Pos");
    uint256 availableLev = _coordinator.getAvailableLeverage();
    require(availableLev >= lvUSDAmount, "Not enough available leverage");
    _burnArchTokenForPosition(msg.sender, archNeededToBurn);
    uint256 positionTokenId = _positionToken.safeMint(userAddress);

    // Checking allownce from an abundence of caution
    if (_ousd.allowance(msg.sender, address(this)) >= ousdPrinciple) {
        _ousd.safeTransferFrom(msg.sender, _addressCoordinator, ousdPrinciple);
    } else {
        uint256 balanceBefore = _ousd.balanceOf(address(this));
        revert("insuff OUSD allowance");
    }

    _coordinator.depositCollateralUnderNFT(positionTokenId, ousdPrinciple);
    _coordinator.getLeveragedOUSD(positionTokenId, lvUSDAmount);

    uint256 positionLeveragedOUSD = ICDP(_addressCDP).getOUSDTotalWithoutInterest(
        ↪ positionTokenId) - ousdPrinciple;
```

```
        require(positionLeveragedOUSD >= minLeverageAmount, "Not enough leveragedOUSD");

        uint256 positionExpireTime = _coordinator.getPositionExpireTime(positionTokenId);

        emit PositionCreated(userAddress, positionTokenId, ousdPrinciple, lvUSDAmount,
            ↪ archNeededToBurn, positionExpireTime);

        return positionTokenId;
    }
```

9.1 ***unwindLeveragedPosition***(*positionTokenId*, *minReturnedOUSD*) X *nonReentrant* whenNotPaused

```
    /// @dev deposit OUSD under NFT ID
    ///
    /// De-leverage and unwind. Send OUSD to msg.sender
    /// must check that the msg.sender owns the NFT
    /// provide msg.sender address to coordinator destroy position
    ///
    /// @param positionTokenId the NFT ID of the position
    function unwindLeveragedPosition(uint256 positionTokenId, uint256 minReturnedOUSD)
        ↪ external nonReentrant whenNotPaused {
        require(_positionToken.ownerOf(positionTokenId) == msg.sender, "Caller is not token
            ↪ owner");
        _positionToken.burn(positionTokenId);
        uint256 positionWindfall = _coordinator.unwindLeveragedOUSD(positionTokenId, msg.
            ↪ sender);
        require(positionWindfall >= minReturnedOUSD, "Not enough OUSD returned");
        emit PositionUnwind(msg.sender, positionTokenId, positionWindfall);
    }
```

9.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

9.1 ***initialize***() X initializer

```
    function initialize() public initializer {
        __Pausable_init();
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
    }
```

9.1 _burnArchTokenForPosition(sender, archAmount)

```
// required - the caller must have allowance for accounts's tokens of at least amount.
function _burnArchTokenForPosition(address sender, uint256 archAmount) internal {
    address treasuryAddress = _parameterStore.getTreasuryAddress();
    _archToken.transferFrom(sender, treasuryAddress, archAmount);
}
```

9.1 _authorizeUpgrade(newImplementation)

```
// solhint-disable-next-line
function _authorizeUpgrade(address newImplementation) internal override {
    _requireAdmin();
}
```

9.1 pauseContract() X onlyGuardian

```
function pauseContract() external onlyGuardian {
    _pause();
}
```

9.1 unPauseContract() X onlyGuardian

```
function unPauseContract() external onlyGuardian {
    _unpause();
}
```

9.1 fallback() X

```
fallback() external {
    revert("LevEngine : Invalid access");
}
```

Chapter 10

Zapper

10.1 contract Zapper

```
contract Zapper is AccessController, ReentrancyGuardUpgradeable, UUPSUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    ICurveFiCurve internal _poolOUSD3CRV;
    IUniswapV2Router02 internal _uniswapRouter;
    IERC20Upgradeable internal _ousd;
    IERC20Upgradeable internal _usdt;
    IERC20Upgradeable internal _usdc;
    IERC20Upgradeable internal _dai;
    LeverageEngine internal _levEngine;
    IERC20Upgradeable internal _archToken;
    ParameterStore internal _paramStore;

    // positionID, // Position ID of the position NFT
    // totalStableAmount, // Total amount of user stable coin zapped in
    // address baseStableAddress, // Base stable address of the stable coin contract
    // bool usedUserArch // Bool representing if user's Arch was used or not

    event ZapIn(uint256 positionID, uint256 totalStableAmount, address baseStableAddress,
        ↪ bool usedUserArch);

    /**************************************************************
    Coin management methods
    **************************************************************/

    address internal constant _ADDRESS_USDT = 0xdAC17F958D2ee523a2206206994597C13D831ec7;
    address internal constant _ADDRESS_USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address internal constant _ADDRESS_DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
    address internal constant _ADDRESS_WETH9 = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address internal constant _ADDRESS_OUSD = 0x2A8e1E676Ec238d8A992307B495b45B3fEAa5e86;
    address internal constant _ADDRESS_3CRV = 0x6c3F90f043a72FA612cbac8115EE7e52BDe6E490;
    address internal constant _ADDRESS_OUSD3CRV_POOL = 0
        ↪ x87650D7bbfC3A9F10587d7778206671719d9910D;
    address internal constant _ADDRESS_UNISWAP_ROUTER = 0
        ↪ x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;
    int128 internal constant _OUSD_TOKEN_INDEX = 0;
}
```

10.1 **zapIn(stableCoinAmount, *cycles*, *archMinAmount*, *ousdMinAmount*, maxSlippageAllowed, *addressBaseStable*, *useUserArch*)** X

```
    /*
        @dev Exchange base stable to OUSD and Arch and create position

        @param stableCoinAmount Amount of stable coin to zap(exchange) into Arch and OUSD
        @param cycles Number of cycles for open position call (determine how much lvUSD will
            ↪ be borrowed)
        @param archMinAmount Minimum amount of Arch tokens to buy
        @param ousdMinAmount Minimum amount of OUSD to buy
        @param maxSlippageAllowed Max slippage allowed in basis points (1/1000). 1000 = 100%
        @param addressBaseStable Address of base stable coin to use for zap
        @param useUserArch If true, will use user arch tokens to open position. If false,
            ↪ will buy arch tokens
    */
    function zapIn(
        uint256 stableCoinAmount,
        uint256 cycles,
        uint256 archMinAmount,
        uint256 ousdMinAmount,
        uint16 maxSlippageAllowed,
        address addressBaseStable,
        bool useUserArch
    ) external returns (uint256) {
        // Whats needs to happen?
        // -1) validate input
        // 0) transfer funds from user to this address
        // 1) figure out how much of stable goes to collateral and how much to pay as arch
            ↪ tokens
        // 2) exchange stable for Arch/ Take from user wallet
        // 3) exchange stable for OUSD
        // 4) open position
        // 5) return NFT to user

        // get a base line of how much stable is under management on conract - should be zero
            ↪  but creating a new base line
        /// validate input
        require(stableCoinAmount > 0, "err:stableCoinAmount==0");
        require(maxSlippageAllowed < 1000, "err:slippage>999");
        require(maxSlippageAllowed > 959, "err:slippage<960");

        // Now we apply slippage. We reduce the min of OUSD
        // This is because we need to always have enough Arch to pay so better to have a bit
            ↪ less OUSD and more Arch than
        // the other way around
        ousdMinAmount = (ousdMinAmount * maxSlippageAllowed) / 1000;

        /// transfer base stable coin from user to this address
        _transferFromSender(addressBaseStable, stableCoinAmount);

        /// Setup
        address[] memory path = _getPath(addressBaseStable);
        uint256 collateralInBaseStableAmount = stableCoinAmount;
        uint256 ousdAmount;

        if (useUserArch == false) {
            // Need to buy Arch tokens. We already know how much Arch tokens we want. We
                ↪ still need to know the Max in stable that
            // we are willing to pay. For that, we're running the splitEstimate again and
                ↪ adding a small buffer
            uint256 coinsToPayForArchAmount;
            (collateralInBaseStableAmount, coinsToPayForArchAmount) = _splitStableCoinAmount(
                ↪ stableCoinAmount, cycles, path, addressBaseStable);
```

```
            /// since we basivally add a buffer for max stable to take, its actually a built
                ↪ in limit on how much slippage is allowed.
            /// In this case up to 5%
            uint256 maxStableToPayForArch = (coinsToPayForArchAmount * 1000) /
                ↪ maxSlippageAllowed;
            // Now swap exact archMinAmount for a maximum of maxStableToPayForArch in stable
                ↪ coin
            uint256 stableUsedForArch = _uniswapRouter.swapTokensForExactTokens(
                archMinAmount,
                maxStableToPayForArch,
                path,
                address(this),
                block.timestamp + 1 minutes
            )[0];

            /// Exchange OUSD from any of the 3CRV. Will revert if didn't get min amount sent
                ↪  (2nd parameter)
            // Now spend all the remainign stable to buy OUSD
            ousdAmount = _exchangeToOUSD(stableCoinAmount - stableUsedForArch, ousdMinAmount,
                ↪  addressBaseStable);
        }

        // Check if we are using existing arch tokens owned by user or buying new ones
        if (useUserArch == true) {
            // First, exchange ALL stable coin to OUSD
            ousdAmount = _exchangeToOUSD(stableCoinAmount, ousdMinAmount, addressBaseStable);
            // We are using owners arch tokens, transfer from msg.sender to address(this)
            uint256 archToTransfer = _getArchAmountToTransferFromUser(ousdAmount, cycles);
            require(_archToken.balanceOf(msg.sender) >= archToTransfer, "err:insuf user arch"
                ↪ );
            require(_archToken.allowance(msg.sender, address(this)) >= archToTransfer, "err:
                ↪ insuf approval arch");
            _transferFromSender(address(_archToken), archToTransfer);
        }

        // calculate min position leverage allowed
        uint256 minLeverageOUSD = (_paramStore.getAllowedLeverageForPosition(ousdAmount,
            ↪ cycles) * maxSlippageAllowed) / 1000;
        // create position
        uint256 tokenId = _levEngine.createLeveragedPositionFromZapper(
            ousdAmount,
            cycles,
            _archToken.balanceOf(address(this)),
            msg.sender,
            minLeverageOUSD
        );

        /// Return all remaining dust/tokens to user
        _archToken.safeTransfer(msg.sender, _archToken.balanceOf(address(this)));

        emit ZapIn(tokenId, stableCoinAmount, addressBaseStable, useUserArch);

        return tokenId;
    }
```

10.1 *previewZapInAmount*(**stableCoinAmount**, *cycles*, ***addressBaseStable***, *useUserArch*)

```
    /*
        @dev simulate OUSD and Arch tokens that will be returned from zapIn call

        @param stableCoinAmount Amount of stable coin to zap(exchange) into Arch and OUSD
        @param cycles Number of cycles for open position call (determine how much lvUSD will
            ↪ borrowed)
        @param addressBaseStable Address of base stable coin to use for zap
        @param useUserArch If true, will use user arch tokens to open position. If false,
            ↪ will buy arch tokens
    */
    function previewZapInAmount(
        uint256 stableCoinAmount,
        uint256 cycles,
        address addressBaseStable,
        bool useUserArch
    ) external view returns (uint256 ousdCollateralAmountReturn, uint256
        ↪ archTokenAmountReturn) {
        /// Setup
        uint256 ousdCollateralAmount;
        uint256 archTokenAmount;

        address[] memory path = _getPath(addressBaseStable);
        int128 stableTokenIndex = _getTokenIndex(addressBaseStable);
        uint256 collateralInBaseStableAmount = stableCoinAmount;

        if (useUserArch == false) {
            // Need to buy Arch tokens. We need to split the stable amount between what we'll
                ↪  as collateral what we'll use to buy Arch
            uint256 coinsToPayForArchAmount;
            (collateralInBaseStableAmount, coinsToPayForArchAmount) = _splitStableCoinAmount(
                ↪ stableCoinAmount, cycles, path, addressBaseStable);
            // preview buy arch tokens from uniswap. results from this will be used as
                ↪ mimimum for Arch to get
            if (addressBaseStable == _ADDRESS_USDC) {
                archTokenAmount = _uniswapRouter.getAmountsOut(coinsToPayForArchAmount, path)
                    ↪ [1];
            } else {
                archTokenAmount = _uniswapRouter.getAmountsOut(coinsToPayForArchAmount, path)
                    ↪ [2];
            }
        }

        // estimate exchange with curve pool
        ousdCollateralAmount = _poolOUSD3CRV.get_dy_underlying(stableTokenIndex,
            ↪ _OUSD_TOKEN_INDEX, collateralInBaseStableAmount);

        if (useUserArch == true) {
            // We are using owners arch tokens, calculate transfer amount from msg.sender to
                ↪ address(this)
            archTokenAmount = _getArchAmountToTransferFromUser(ousdCollateralAmount, cycles);
        }

        return (ousdCollateralAmount, archTokenAmount);
    }
```

10.1 previewTokenSplit(stableCoinAmount, cycles, addressBaseStable)

```
    /*
        @dev estimate how much of base stable will be used to get Arch and how much will be
            ↪ used to get OUSD
        @param stableCoinAmount Amount of stable coin to zap(exchange) into Arch and OUSD
        @param cycles Number of cycles for open position call (determine how much lvUSD will
            ↪ borrowed)
        @param addressBaseStable Address of base stable coin to use for zap
    */
    function previewTokenSplit(
        uint256 stableCoinAmount,
        uint256 cycles,
        address addressBaseStable
    ) external view returns (uint256 collateralInBaseStableAmount, uint256
        ↪ coinsToPayForArchInStableAmount) {
        address[] memory path = _getPath(addressBaseStable);
        return _splitStableCoinAmount(stableCoinAmount, cycles, path, addressBaseStable);
    }
```

10.1 _calcCollateralBasedOnArchPrice(stableCoinAmount, archPriceInStable, multiplierOfLeverageFromOneCollateral, decimal)

```
    /*****************************************************************
    Split stable coin methods to collateral amount and arch amount
    *****************************************************************/
    function _calcCollateralBasedOnArchPrice(
        uint256 stableCoinAmount,
        uint256 archPriceInStable,
        uint256 multiplierOfLeverageFromOneCollateral,
        uint8 decimal
    ) internal view returns (uint256 collateralAmountReturned) {
        /// TODO: Add comments and explain the formula
        uint256 archToLevRatio = _paramStore.getArchToLevRatio();
        uint256 tempCalc = (multiplierOfLeverageFromOneCollateral * archPriceInStable) / 1
            ↪ ether;
        uint256 ratioOfColl = (archToLevRatio * 10**decimal) / (archToLevRatio + tempCalc *
            ↪ 10**(18 - decimal));
        uint256 collateralAmount = (stableCoinAmount * ratioOfColl) / 10**decimal;
        return collateralAmount;
    }
```

10.1 _getCollateralAmount(stableCoinAmount, cycles, path, decimal)

```
    function _getCollateralAmount(
        uint256 stableCoinAmount,
        uint256 cycles,
        address[] memory path,
        uint8 decimal
    ) internal view returns (uint256) {
        // Calculate how much leverage is needed per 1 OUSD. Used throughout the calculation
            ↪ as a constant multiplier
        uint256 multiplierOfLeverageFromOneCollateral = _paramStore.
            ↪ getAllowedLeverageForPosition(1 ether, cycles);
        // Get the price of 1 Arch in stableCoin.
        uint256 archPriceInStable = _uniswapRouter.getAmountsIn(1 ether, path)[0];
        // calculate first estimation of collateral amount, based on price of a single arch
            ↪ token.
        uint256 collateralAmount = _calcCollateralBasedOnArchPrice(
            stableCoinAmount,
            archPriceInStable,
            multiplierOfLeverageFromOneCollateral,
            decimal
```

```
        );

        // Now we have an estimate of how much collateral have, so we can calc how much Arch
            ↪ we need
        // Do a second round of calc where everything is the same, just with the Arch price
            ↪ being more accurate
        uint256 collateralAmountIn18Decimal = collateralAmount * 10**(18 - decimal);
        uint256 archAmountEstimated = _paramStore.calculateArchNeededForLeverage(
            ((collateralAmountIn18Decimal) * multiplierOfLeverageFromOneCollateral) / 1 ether
        );
        // Now  that we know how much arch we are going to need to get, we can use Uniswap
            ↪ amountIn method to estimate
        // the actual (much better estimated then before) price in stable coin of 1 Arch
            ↪ token
        archPriceInStable = ((_uniswapRouter.getAmountsIn(archAmountEstimated, path)[0] * 1
            ↪ ether) / archAmountEstimated);
        collateralAmount = _calcCollateralBasedOnArchPrice(stableCoinAmount,
            ↪ archPriceInStable, multiplierOfLeverageFromOneCollateral, decimal);
        return collateralAmount;
    }
```

10.1 _splitStableCoinAmount(stableCoinAmount, cycles, path, addressStable)

```
    // TODO: pass it the max slippege allowed for line 196
    function _splitStableCoinAmount(
        uint256 stableCoinAmount,
        uint256 cycles,
        address[] memory path,
        address addressStable
    ) internal view returns (uint256 stableForCollateral, uint256 stableForArch) {
        uint8 decimal = _getTokenDecimal(addressStable);
        // Figure out how much of stable goes to OUSD and how much to pay as arch tokens
        uint256 collateralInBaseStableAmount = _getCollateralAmount(stableCoinAmount, cycles,
            ↪  path, decimal);
        // Set aside a bit less for collateral, to reduce risk of revert
        collateralInBaseStableAmount = (collateralInBaseStableAmount * 999) / 1000;
        uint256 coinsToPayForArchAmount = stableCoinAmount - collateralInBaseStableAmount;
        return (collateralInBaseStableAmount, coinsToPayForArchAmount);
    }
```

10.1 _getArchAmountToTransferFromUser(ousdAmount, cycles)

```
    /**************************************************************
     * transfer methods
     *************************************************************/

    function _getArchAmountToTransferFromUser(uint256 ousdAmount, uint256 cycles) internal
        ↪ view returns (uint256) {
        return _paramStore.calculateArchNeededForLeverage(_paramStore.
            ↪ getAllowedLeverageForPosition(ousdAmount, cycles));
    }
```

10.1 _transferFromSender(tokenAddress, amount)

```
    function _transferFromSender(address tokenAddress, uint256 amount) internal {
        IERC20Upgradeable(tokenAddress).safeTransferFrom(msg.sender, address(this), amount);
    }
```

10.1 _exchangeToOUSD(amount, minAmountToReceive, addressBaseStable)

```
    function _exchangeToOUSD(
        uint256 amount,
        uint256 minAmountToReceive,
        address addressBaseStable
    ) internal returns (uint256 amountOUSDReceived) {
        IERC20Upgradeable(addressBaseStable).safeApprove(address(_poolOUSD3CRV), amount);
        int128 fromTokenIndex = _getTokenIndex(addressBaseStable);
        uint256 amountReceived = _poolOUSD3CRV.exchange_underlying(fromTokenIndex,
            ↪ _OUSD_TOKEN_INDEX, amount, minAmountToReceive);
        IERC20Upgradeable(addressBaseStable).safeApprove(address(_poolOUSD3CRV), 0);
        return amountReceived;
    }
```

10.1 _getPath(addressBaseStable)

```
    /// Coin 0 in pool is OUSD
    /// Coin 1 in pool is DAI
    /// Coin 2 in pool is USDC
    /// Coin 3 in pool is USDT
    /// using https://etherscan.io/address/0xB9fC157394Af804a3578134A6585C0dc9cc990d4#
        ↪ readContract
    /// On pool OUSD pool 0x87650D7bbfC3A9F10587d7778206671719d9910D
    /// CurveIndex, name, address - decimals
    //  [0] OUSD 0x2A8e1E676Ec238d8A992307B495b45B3fEAa5e86 - 18
    //  [1] DAI 0x6B175474E89094C44Da98b954EedeAC495271d0F - 18
    //  [2] USDC 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 - 6
    //  [3] USDT 0xdAC17F958D2ee523a2206206994597C13D831ec7 - 6

    // _getPath determines the Uniswap exchange path
    // There exists a USDC / ARCH Uniswap pool
    // If the user has USDT or DAI we must first convert to USDC
    function _getPath(address addressBaseStable) internal view returns (address[] memory) {
        address[] memory path;
        if (addressBaseStable == _ADDRESS_USDC) {
            // Base stable is already USDC, no conversion needed
            path = new address[](2);
            path[0] = addressBaseStable;
            path[1] = address(_archToken);
        } else {
            // Base stable is not USDC, must convert to USDC first
            path = new address[](3);
            path[0] = addressBaseStable;
            path[1] = _ADDRESS_USDC;
            path[2] = address(_archToken);
        }
        return path;
    }
```

10.1 *_getTokenIndex*(***addressBaseStable***)

```
    function _getTokenIndex(address addressBaseStable) internal pure returns (int128
        ↪ tokenIndex) {
        if (addressBaseStable == _ADDRESS_USDT) {
            return tokenIndex = 3;
        }
        if (addressBaseStable == _ADDRESS_USDC) {
            return tokenIndex = 2;
        }
        if (addressBaseStable == _ADDRESS_DAI) {
            return tokenIndex = 1;
        }
        revert("Zapper: Unsupported stablecoin");
    }
```

10.1 _getTokenDecimal(***addressBaseStable***)

```
    function _getTokenDecimal(address addressBaseStable) internal pure returns (uint8) {
        if (addressBaseStable == _ADDRESS_USDT) {
            return 6;
        }
        if (addressBaseStable == _ADDRESS_USDC) {
            return 6;
        }
        if (addressBaseStable == _ADDRESS_DAI) {
            return 18;
        }
        revert("Zapper: Unsupported stablecoin");
    }
```

10.1 **_authorizeUpgrade**(**newImplementation**)

```
    /**************************************************************
    Admin methods
    **************************************************************/

    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

10.1 ***initialize***() X initializer

```
    function initialize() public initializer {
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
    }
```

10.1 setDependencies(addressLevEngine, addressArchToken, addressParamStore) X nonReentrant onlyAdmin

```
    function setDependencies(
        address addressLevEngine,
        address addressArchToken,
        address addressParamStore
    ) external nonReentrant onlyAdmin {
        // Load contracts
        _ousd = IERC20Upgradeable(_ADDRESS_OUSD);
        _usdt = IERC20Upgradeable(_ADDRESS_USDT);
        _usdc = IERC20Upgradeable(_ADDRESS_USDC);
        _dai = IERC20Upgradeable(_ADDRESS_DAI);
        _poolOUSD3CRV = ICurveFiCurve(_ADDRESS_OUSD3CRV_POOL);
        _uniswapRouter = IUniswapV2Router02(_ADDRESS_UNISWAP_ROUTER);
        _levEngine = LeverageEngine(addressLevEngine);
        _archToken = IERC20Upgradeable(addressArchToken);
        _paramStore = ParameterStore(addressParamStore);

        /// Need to approve for both Arch and ousd
        _ousd.safeApprove(addressLevEngine, 0);
        _ousd.safeApprove(addressLevEngine, type(uint256).max);

        /// Need to approve for both Arch and ousd
        _archToken.safeApprove(addressLevEngine, 0);
        _archToken.safeApprove(addressLevEngine, type(uint256).max);

        _usdt.safeApprove(address(_uniswapRouter), 0);
        _usdt.safeApprove(address(_uniswapRouter), type(uint256).max);

        _usdc.safeApprove(address(_uniswapRouter), 0);
        _usdc.safeApprove(address(_uniswapRouter), type(uint256).max);

        _dai.safeApprove(address(_uniswapRouter), 0);
        _dai.safeApprove(address(_uniswapRouter), type(uint256).max);
    }
```

Chapter 11

Coordinator

11.1 contract *Coordinator*

```
/// @title Coordinator
/// @dev is in charge of overall flow of creating positions and unwinding positions
/// It manages keeping tracks of fund in vault, updating CDP as needed and transferring lvUSD
    ↪  inside the system
/// It is controlled (and called) by the leverage engine
contract Coordinator is ICoordinator, AccessController, ReentrancyGuardUpgradeable,
    ↪ UUPSUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;
    address internal _addressLvUSD;
    address internal _addressVaultOUSD;
    address internal _addressCDP;
    address internal _addressOUSD;
    address internal _addressExchanger;
    address internal _addressPoolManager;
    address internal _addressAuction;

    VaultOUSD internal _vault;
    CDPosition internal _cdp;
    Exchanger internal _exchanger;
    IERC20Upgradeable internal _lvUSD;
    IERC20Upgradeable internal _ousd;
    ParameterStore internal _paramStore;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

11.1 **setDependencies**(addressLvUSD, *addressVaultOUSD*, addressCDP, *addressOUSD*, *addressExchanger*, *addressParamStore*, *addressPoolManager*, addressAuction) X *nonReentrant* *onlyAdmin*

```
    function setDependencies(
        address addressLvUSD,
        address addressVaultOUSD,
        address addressCDP,
        address addressOUSD,
        address addressExchanger,
        address addressParamStore,
        address addressPoolManager,
        address addressAuction
    ) external nonReentrant onlyAdmin {
```

```
        require(addressLvUSD != address(0), "cant set to 0 A");
        require(addressVaultOUSD != address(0), "cant set to 0 A");
        require(addressCDP != address(0), "cant set to 0 A");
        require(addressOUSD != address(0), "cant set to 0 A");
        require(addressExchanger != address(0), "cant set to 0 A");
        require(addressParamStore != address(0), "cant set to 0 A");
        require(addressPoolManager != address(0), "cant set to 0 A");
        require(addressAuction != address(0), "cant set to 0 A");

        _addressLvUSD = addressLvUSD;
        _addressVaultOUSD = addressVaultOUSD;
        _addressCDP = addressCDP;
        _addressOUSD = addressOUSD;
        _addressExchanger = addressExchanger;
        _addressPoolManager = addressPoolManager;
        _addressAuction = addressAuction;

        _vault = VaultOUSD(_addressVaultOUSD);
        _cdp = CDPosition(_addressCDP);
        _exchanger = Exchanger(_addressExchanger);
        _lvUSD = IERC20Upgradeable(_addressLvUSD);
        _ousd = IERC20Upgradeable(_addressOUSD);
        _paramStore = ParameterStore(addressParamStore);

        // /// reset allownce
        _ousd.safeApprove(_addressVaultOUSD, 0);
        _lvUSD.safeApprove(_addressPoolManager, 0);

        // approve VaultOUSD address to spend OUSD on behalf of coordinator
        _ousd.safeApprove(_addressVaultOUSD, type(uint256).max);
        _lvUSD.safeApprove(_addressPoolManager, type(uint256).max);
    }
```

11.1 _coordinatorLvUSDTransferToExchanger(amount)

```
    function _coordinatorLvUSDTransferToExchanger(uint256 amount) internal {
        uint256 currentCoordinatorLeverageBalance = getAvailableLeverage();
        uint256 currentBalanceOnLvUSDContract = _lvUSD.balanceOf(address(this));
        require(currentCoordinatorLeverageBalance >= amount, "insuf levAv to trnsf");
        require(currentBalanceOnLvUSDContract >= amount, "insuf lvUSD balance to trnsf");

        _paramStore.changeCoordinatorLeverageBalance(currentCoordinatorLeverageBalance -
            ↪ amount);
        _lvUSD.safeTransfer(_addressExchanger, amount);
    }
```

11.1 acceptLeverageAmount(leverageAmountToAccept) X onlyAuctioneer nonReentrant

```
    function acceptLeverageAmount(uint256 leverageAmountToAccept) external onlyAuctioneer
        ↪ nonReentrant {
        require(Auction(_addressAuction).isAuctionClosed() == false, "Auction must be open");
        uint256 currentLvUSDBalance = _lvUSD.balanceOf(address(this));
        require(currentLvUSDBalance >= leverageAmountToAccept, "lvUSD !< levAmt");
        _paramStore.changeCoordinatorLeverageBalance(leverageAmountToAccept);
    }
```

11.1 resetAndBurnLeverage() X onlyAdmin nonReentrant

```
    function resetAndBurnLeverage() external onlyAdmin nonReentrant {
        uint256 coordinatorCurrentLvUSDBalance = _lvUSD.balanceOf(address(this));
        ERC20Burnable(_addressLvUSD).burn(coordinatorCurrentLvUSDBalance);
        _paramStore.changeCoordinatorLeverageBalance(0);
    }
```

11.1 depositCollateralUnderNFT(_nftId, _amountInOUSD) X nonReentrant onlyExecutive

```
    /* Privileged functions: Executive */

    // Note: Expects funds to be under coordinator already
    function depositCollateralUnderNFT(uint256 _nftId, uint256 _amountInOUSD) external
        ↪ override nonReentrant onlyExecutive {
        /// Transfer collateral to vault, mint shares to shares owner
        uint256 shares = _vault.archimedesDeposit(_amountInOUSD, address(this));
        // create CDP position with collateral
        _cdp.createPosition(_nftId, _amountInOUSD);
        _cdp.addSharesToPosition(_nftId, shares);
    }
```

11.1 borrowUnderNFT(_nftId, _amount) X nonReentrant onlyExecutive

```
    function borrowUnderNFT(uint256 _nftId, uint256 _amount) external override nonReentrant
        ↪ onlyExecutive {
        _borrowUnderNFT(_nftId, _amount);
    }
```

11.1 repayUnderNFT(_nftId, _amountLvUSDToRepay) X nonReentrant onlyExecutive

```
    function repayUnderNFT(uint256 _nftId, uint256 _amountLvUSDToRepay) external override
        ↪ nonReentrant onlyExecutive {
        _repayUnderNFT(_nftId, _amountLvUSDToRepay);
    }
```

11.1 getLeveragedOUSD(_nftId, _amountToLeverage) X nonReentrant onlyExecutive

```
    function getLeveragedOUSD(uint256 _nftId, uint256 _amountToLeverage) external override
        ↪ nonReentrant onlyExecutive {
        /* Flow
          1. basic sanity checks
          2. borrow lvUSD
          3. call exchanger to exchange lvUSD. Exchanged OUSD will be under Coordinator
              ↪ address.  Save exchanged OUSD value
          4. deposit OUSD funds in Vault
          5. Update CDP totalOUSD and shares for nft position
        */

        uint256 ousdPrinciple = _cdp.getOUSDPrinciple(_nftId);
        require(
            _amountToLeverage <= _paramStore.getAllowedLeverageForPosition(ousdPrinciple,
                ↪ _paramStore.getMaxNumberOfCycles()),
            "Leverage more than max allowed"
        );
        // borrowUnderNFT transfer lvUSD from Coordinator to Exchanger + mark borrowed lvUSD
            ↪ in CDP under nft ID
        _borrowUnderNFT(_nftId, _amountToLeverage);

        uint256 ousdAmountExchanged = _exchanger.swapLvUSDforOUSD(_amountToLeverage);
```

```
        uint256 feeTaken = _takeOriginationFee(ousdAmountExchanged);
        uint256 positionLeveragedOUSDAfterFees = ousdAmountExchanged - feeTaken;
        uint256 sharesFromDeposit = _vault.archimedesDeposit(positionLeveragedOUSDAfterFees,
            ↪ address(this));

        _cdp.addSharesToPosition(_nftId, sharesFromDeposit);
        _cdp.depositOUSDtoPosition(_nftId, positionLeveragedOUSDAfterFees);
    }
```

11.1 unwindLeveragedOUSD(_nftId, _userAddress) X nonReentrant onlyExecutive

```
    function unwindLeveragedOUSD(uint256 _nftId, address _userAddress)
        external
        override
        nonReentrant
        onlyExecutive
        returns (uint256 positionWindfall)
    {
        /* Flow
            1. sanity checks as needed
            2. get amount of shares for position
            3. redeem shares for OUSD (from vault), OUSD is assigned to exchanger
            4. exchange as much OUSD as needed to cover lvUSD debt (do we want to exchange
                ↪ principle as well?)
                    // slippage of 0.2% is ok, if dont - revert!
            5. repay lvUSD
            6. return what OUSD is left to _userAddress
            7. delete CDP position
        */

        uint256 numberOfSharesInPosition = _cdp.getShares(_nftId);
        uint256 borrowedLvUSD = _cdp.getLvUSDBorrowed(_nftId);
        require(numberOfSharesInPosition != 0, "Position has no shares");

        uint256 redeemedOUSD = _vault.archimedesRedeem(numberOfSharesInPosition,
            ↪ _addressExchanger, address(this));

        (uint256 exchangedLvUSD, uint256 remainingOUSD) = _exchanger.swapOUSDforLvUSD(
            ↪ redeemedOUSD, borrowedLvUSD);

        _repayUnderNFT(_nftId, exchangedLvUSD);

        // transferring funds from coordinator to user
        _ousd.safeTransfer(_userAddress, remainingOUSD);

        /// Note : leverage engine still need to make sure the delete the NFT itself in
            ↪ positionToken
        _cdp.deletePosition(_nftId);

        return remainingOUSD;
    }
```

11.1 getAvailableLeverage()

```
    /* Privileged functions: Anyone */

    function getAvailableLeverage() public view returns (uint256) {
        return _paramStore.getCoordinatorLeverageBalance();
    }
```

11.1 getPositionExpireTime(_nftId)

```
    function getPositionExpireTime(uint256 _nftId) external view override returns (uint256) {
        return _cdp.getPositionExpireTime(_nftId);
    }
```

11.1 addressOfLvUSDToken()

```
    function addressOfLvUSDToken() external view override returns (address) {
        return _addressLvUSD;
    }
```

11.1 addressOfVaultOUSDToken()

```
    function addressOfVaultOUSDToken() external view override returns (address) {
        return _addressVaultOUSD;
    }
```

11.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

11.1 initialize() X initializer

```
    function initialize() public initializer {
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());
        setAuctioneer(_msgSender());
    }
```

11.1 *_borrowUnderNFT*(*_nftId*, _amount)

```
    function _borrowUnderNFT(uint256 _nftId, uint256 _amount) internal {
        _coordinatorLvUSDTransferToExchanger(_amount);
        _cdp.borrowLvUSDFromPosition(_nftId, _amount);
    }
```

11.1 ***_repayUnderNFT***(*_nftId*, ***_amountLvUSDToRepay***)

```
    function _repayUnderNFT(uint256 _nftId, uint256 _amountLvUSDToRepay) internal {
        _cdp.repayLvUSDToPosition(_nftId, _amountLvUSDToRepay);
    }
```

11.1 *_takeOriginationFee*(**_leveragedOUSDAmount**)

```
    function _takeOriginationFee(uint256 _leveragedOUSDAmount) internal returns (uint256 fee)
        ↪  {
        uint256 _fee = _paramStore.calculateOriginationFee(_leveragedOUSDAmount);
        _ousd.safeTransfer(_paramStore.getTreasuryAddress(), _fee);
        return _fee;
    }
```

11.1 *_checkEqualBalanceWithBuffer*(***givenAmount***, **expectedAmount**)

```
    function _checkEqualBalanceWithBuffer(uint256 givenAmount, uint256 expectedAmount)
        ↪ internal returns (bool) {
        uint256 expectedLowerBound = expectedAmount - 10; // to accomadte rounding in the 2
            ↪ lowest digits
        uint256 expectedUpperBound = expectedAmount + 10; // to accomadte rounding in the 2
            ↪ lowest digits
        return (expectedLowerBound <= givenAmount) && (givenAmount <= expectedUpperBound);
    }
```

11.1 **_authorizeUpgrade**(**newImplementation**)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

11.1 interface ***ICoordinator***

```
interface ICoordinator {
    /*================INTERFACE SCRATCH AREA=========*/
    // parameters for contract
    // - lvUSD Token contract address
    // - OUSD Vault contract address
    /*===============================================*/
}
```

11.1 *depositCollateralUnderNFT*(*_nftId*, **_amountInOUSD**) X [***ICoordinator***]

```
    /* Privileged functions: Executive */

    /// @dev deposit OUSD under NFT ID
    ///
    /// User sends OUSD to the contract. OUSD is written under NFT ID
    ///
    /// @param _nftId the Archimedes ERC-721 token id
    /// @param _amountInOUSD the amount of OUSD sent to Archimedes
    function depositCollateralUnderNFT(uint256 _nftId, uint256 _amountInOUSD) external;
```

11.1 borrowUnderNFT(*_nftId*, **_amountLvUSDToBorrow**) X [***ICoordinator***]

```
    /// @dev Borrow lvUSD under NFT ID
    ///
    /// User borrow lvUSD against the OUSD deposited as collateral in Vault
    /// Need to check collaterallization ratio
    /// Need to collect origination fee and sent them to vault
    ///
    /// @param _amountLvUSDToBorrow the amount of lvUSD requested
    /// @param _nftId the Archimedes ERC-721 token id
    function borrowUnderNFT(uint256 _nftId, uint256 _amountLvUSDToBorrow) external;
```

11.1 *repayUnderNFT*(*_nftId*, ***_amountLvUSDToRepay***) X [***ICoordinator***]

```
    /// @dev Repay lvUSD under NFT ID
    ///
    /// User repay lvUSD against the OUSD deposited as collateral
    /// Need to check collaterallization ratio
    ///
    /// @param _amountLvUSDToRepay the amount of lvUSD requested
    /// @param _nftId the Archimedes ERC-721 token id
    function repayUnderNFT(uint256 _nftId, uint256 _amountLvUSDToRepay) external;
```

11.1 **getLeveragedOUSD**(*_nftId*, _amountToLeverage) X [***ICoordinator***]

```
    /// @dev borrow lvUSD and exchange it for OUSD
    /// @param _nftId NFT ID
    /// @param _amountToLeverage amount to borrow
    function getLeveragedOUSD(uint256 _nftId, uint256 _amountToLeverage) external;
```

11.1 unwindLeveragedOUSD(*_nftId*, *_userAddress*) X [***ICoordinator***]

```
    /// @dev unwind position by repaying lvUSD debt using existing OUSD funds in position
    /// @param _nftId NFT ID
    /// @param _userAddress address to transfer leftover OUSD to
    function unwindLeveragedOUSD(uint256 _nftId, address _userAddress) external returns (
        ↪ uint256 positionWindfall);
```

11.1 ***addressOfLvUSDToken***() X [***ICoordinator***]

```
    /// @dev returns the address of lvUSD contract on file
    function addressOfLvUSDToken() external returns (address);
```

11.1 addressOfVaultOUSDToken() X [***ICoordinator***]

```
    /// @dev returns the address of VaultOUSD contract on file
    function addressOfVaultOUSDToken() external returns (address);
```

11.1 **getAvailableLeverage**() [***ICoordinator***]

```
    function getAvailableLeverage() external view returns (uint256);
```

11.1 ***getPositionExpireTime***(*_nftId*) [***ICoordinator***]

```
    /// @dev callthrough to CDP to get expiration timestamp
    /// @param _nftId NFT ID
    function getPositionExpireTime(uint256 _nftId) external view returns (uint256);
```

Chapter 12

VaultOUSD

12.1 contract *VaultOUSD*

```
/// @title Archimedes OUSD vault
/// @notice Vault holds OUSD managed by Archimedes under all positions.
/// @notice It Uses ER4626 to mint shares for deposited OUSD.
contract VaultOUSD is ERC4626Upgradeable, AccessController, ReentrancyGuardUpgradeable,
    ↪ UUPSUpgradeable {
    using SafeERC20Upgradeable for IERC20Upgradeable;

    ParameterStore internal _paramStore;
    IOUSD internal _ousd;

    uint256 internal _assetsHandledByArchimedes;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

12.1 *fallback*() X

```
    fallback() external {
        revert("VaultOUSD : Invalid access");
    }
```

12.1 **setDependencies**(_addressParamStore, *_addressOUSD*) X *onlyAdmin*

```
    function setDependencies(address _addressParamStore, address _addressOUSD) external
        ↪ onlyAdmin {
        require(_addressParamStore != address(0), "cant set to 0 A");
        require(_addressOUSD != address(0), "cant set to 0 A");

        _paramStore = ParameterStore(_addressParamStore);
        _ousd = IOUSD(_addressOUSD);
        _optInForRebases();
    }
```

12.1 *archimedesDeposit*(*assets*, *receiver*) X *nonReentrant* *onlyExecutive*

```
    function archimedesDeposit(uint256 assets, address receiver) external nonReentrant
        ↪ onlyExecutive returns (uint256) {
        _takeRebaseFees();
        _assetsHandledByArchimedes += assets;
        return super.deposit(assets, receiver);
    }
```

12.1 redeem(shares, *receiver*, owner) X

```
    function redeem(
        uint256 shares,
        address receiver,
        address owner
    ) public virtual override returns (uint256) {
        revert("call ArchimedesRedeem instead");
    }
```

12.1 *deposit*(*assets*, *receiver*) X

```
    function deposit(uint256 assets, address receiver) public virtual override returns (
        ↪ uint256) {
        revert("call ArchimedesDeposit instead");
    }
```

12.1 mint(shares, *receiver*) X

```
    function mint(uint256 shares, address receiver) public virtual override returns (uint256)
        ↪  {
        revert("cant mint on vault");
    }
```

12.1 *withdraw*(*assets*, *receiver*, owner) X

```
    function withdraw(
        uint256 assets,
        address receiver,
        address owner
    ) public virtual override returns (uint256) {
        revert("cant withdraw on vault");
    }
```

12.1 *archimedesRedeem*(shares, *receiver*, owner) X *nonReentrant* *onlyExecutive*

```
    function archimedesRedeem(
        uint256 shares,
        address receiver,
        address owner
    ) external nonReentrant onlyExecutive returns (uint256) {
        _takeRebaseFees();
        uint256 redeemedAmountInAssets = super.redeem(shares, receiver, owner);
        /// This is due to integer rounding issues. If this is the case, reset
            ↪ _assetsHandledByArchimedes
        if (_assetsHandledByArchimedes < redeemedAmountInAssets) {
            _assetsHandledByArchimedes = 0;
        } else {
            _assetsHandledByArchimedes = _assetsHandledByArchimedes - redeemedAmountInAssets;
```

```
        }
        return redeemedAmountInAssets;
    }
```

12.1 takeRebaseFees() X nonReentrant onlyAdmin

```
    function takeRebaseFees() external nonReentrant onlyAdmin {
        _takeRebaseFees();
    }
```

12.1 _optInForRebases()

```
    function _optInForRebases() internal {
        _ousd.rebaseOptIn();
    }
```

12.1 constructor() X

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

12.1 initialize(asset, name, symbol) X initializer

```
    function initialize(
        IERC20MetadataUpgradeable asset,
        string memory name,
        string memory symbol
    ) public initializer {
        __AccessControl_init();
        __ReentrancyGuard_init();
        __UUPSUpgradeable_init();

        _grantRole(ADMIN_ROLE, _msgSender());
        setGovernor(_msgSender());
        setExecutive(_msgSender());
        setGuardian(_msgSender());

        __ERC4626_init(asset);
        __ERC20_init(name, symbol);

        _assetsHandledByArchimedes = 0;
    }
```

12.1 *_takeRebaseFees*()

```
    function _takeRebaseFees() internal {
        uint256 roundingBuffer = 100; // wei

        // If for some reason, _assetsHandledByArchimedes is larger then total assets, reset
            ↪ _assetsHandledByArchimedes to max (ie total assets)
        uint256 totalAssetsCurrent = totalAssets();
        if (totalAssetsCurrent < _assetsHandledByArchimedes) {
            if (_assetsHandledByArchimedes - totalAssetsCurrent > 1000) {
                revert("Err:ArchAssets > totalA");
            }
            // This is due to drifting in handling assets. reset drift
            // console.log("reseting drift in vault _assetsHandledByArchimedes %s, total
                ↪ assets %s", _assetsHandledByArchimedes, totalAssetsCurrent);
            _assetsHandledByArchimedes = totalAssetsCurrent;
        }

        // Another layer of securing from rounding errors - round down last 2 digits if
            ↪ possible
        uint256 unhandledRebasePayment;
        if ((totalAssets() - _assetsHandledByArchimedes) > 100) {
            unhandledRebasePayment = ((totalAssets() - _assetsHandledByArchimedes) / 100) *
                ↪ 100;
        } else {
            unhandledRebasePayment = 0;
        }

        /// only run fee collection if there are some rebased funds not handled (pad by
            ↪ rounding buffer)
        if (unhandledRebasePayment > roundingBuffer) {
            uint256 feeToCollect = (unhandledRebasePayment * _paramStore.getRebaseFeeRate())
                ↪ / 1 ether;
            uint256 handledRebaseValueToKeepInVault = unhandledRebasePayment - feeToCollect;

            _assetsHandledByArchimedes += handledRebaseValueToKeepInVault;

            _ousd.transfer(_paramStore.getTreasuryAddress(), feeToCollect);
        }
    }
```

12.1 **_authorizeUpgrade**(**newImplementation**)

```
    // solhint-disable-next-line
    function _authorizeUpgrade(address newImplementation) internal override {
        _requireAdmin();
    }
```

Chapter 13

Dependencies

13.1 contract *BasicAccessController*

```
abstract contract BasicAccessController is AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    address private _addressMinter;

    address private _nominatedAdmin;
    address private _oldAdmin;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */
    uint256[44] private __gap;
}
```

13.1 modifier *onlyAdmin*()

```
    modifier onlyAdmin() {
        require(hasRole(ADMIN_ROLE, msg.sender), "Caller is not Admin");
        _;
    }
```

13.1 modifier **onlyMinter()**

```
    modifier onlyMinter() {
        require(hasRole(MINTER_ROLE, msg.sender), "Caller is not Minter");
        _;
    }
```

13.1 setAdmin(*newAdmin*) X ***onlyAdmin***

```
    function setAdmin(address newAdmin) public onlyAdmin {
        if (newAdmin == _msgSender()) {
            revert("new admin must be different");
        }
        _nominatedAdmin = newAdmin;
        _oldAdmin = _msgSender();
    }
```

13.1 ***acceptAdminRole***() X

```
    function acceptAdminRole() external {
        if (_nominatedAdmin == address(0) || _oldAdmin == address(0)) {
            revert("no nominated admin");
        }
        if (_nominatedAdmin == _msgSender()) {
            _grantRole(ADMIN_ROLE, _msgSender());
            _revokeRole(ADMIN_ROLE, _oldAdmin);

            _nominatedAdmin = address(0);
            _oldAdmin = address(0);
        }
    }
```

13.1 ***renounceRole***(*role*, account) X

```
    function renounceRole(bytes32 role, address account) public virtual override {
        if (hasRole(ADMIN_ROLE, msg.sender)) {
            revert("Admin cant use renounceRole");
        }
        require(account == _msgSender(), "can only renounce roles for self");

        _revokeRole(role, account);
    }
```

13.1 **setMinter**(*newMinter*) X ***onlyAdmin***

```
    function setMinter(address newMinter) public onlyAdmin {
        address oldMinter = _addressMinter;
        require(oldMinter != newMinter, "New minter must be different");
        _grantRole(MINTER_ROLE, newMinter);
        _revokeRole(MINTER_ROLE, oldMinter);
        _addressMinter = newMinter;
    }
```

13.1 *getAddressMinter*()

```
    function getAddressMinter() public view returns (address) {
        return _addressMinter;
    }
```

13.1 ***_requireAdmin***()

```
    function _requireAdmin() internal view {
        require(hasRole(ADMIN_ROLE, msg.sender), "Caller is not admin");
    }
```

13.2 contract AccessController

```
/// @title ArchRole
/// @dev Contract used to inherit standard role enforcement across Archimedes contracts
abstract contract AccessController is AccessControlUpgradeable {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant EXECUTIVE_ROLE = keccak256("EXECUTIVE_ROLE");
    bytes32 public constant GOVERNOR_ROLE = keccak256("GOVERNOR_ROLE");
    bytes32 public constant GUARDIAN_ROLE = keccak256("GUARDIAN_ROLE");
    bytes32 public constant AUCTIONEER = keccak256("AUCTIONEER");

    address internal _addressAuctioneer;
    address private _addressExecutive;
    address private _addressGovernor;
    address private _addressGuardian;

    address private _nominatedAdmin;
    address private _oldAdmin;

    /**
     * @dev This empty reserved space is put in place to allow future versions to add new
     * variables without shifting down storage in the inheritance chain.
     * See https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps
     */

    uint256[44] private __gap;
}
```

13.2 modifier onlyAdmin()

```
    modifier onlyAdmin() {
        require(hasRole(ADMIN_ROLE, msg.sender), "Caller is not Admin");
        _;
    }
```

13.2 modifier onlyExecutive()

```
    modifier onlyExecutive() {
        require(hasRole(EXECUTIVE_ROLE, msg.sender), "Caller is not Executive");
        _;
    }
```

13.2 modifier onlyGovernor()

```
    modifier onlyGovernor() {
        require(hasRole(GOVERNOR_ROLE, msg.sender), "Caller is not Governor");
        _;
    }
```

13.2 modifier *onlyGuardian*()

```
    modifier onlyGuardian() {
        require(hasRole(GUARDIAN_ROLE, msg.sender), "Caller is not Guardian");
        _;
    }
```

13.2 modifier ***onlyAuctioneer***()

```
    modifier onlyAuctioneer() {
        require(hasRole(AUCTIONEER, msg.sender), "Caller is not Auctioneer");
        _;
    }
```

13.2 setAdmin(*newAdmin*) X ***onlyAdmin***

```
    function setAdmin(address newAdmin) public onlyAdmin {
        if (newAdmin == _msgSender()) {
            revert("new admin must be different");
        }
        _nominatedAdmin = newAdmin;
        _oldAdmin = _msgSender();
    }
```

13.2 ***acceptAdminRole***() X

```
    function acceptAdminRole() external {
        if (_nominatedAdmin == address(0) || _oldAdmin == address(0)) {
            revert("no nominated admin");
        }
        if (_nominatedAdmin == _msgSender()) {
            _grantRole(ADMIN_ROLE, _msgSender());
            _revokeRole(ADMIN_ROLE, _oldAdmin);

            _nominatedAdmin = address(0);
            _oldAdmin = address(0);
        }
    }
```

13.2 ***renounceRole***(*role*, account) X

```
    function renounceRole(bytes32 role, address account) public virtual override {
        if (hasRole(ADMIN_ROLE, msg.sender)) {
            revert("Admin cant use renounceRole");
        }
        require(account == _msgSender(), "can only renounce roles for self");

        _revokeRole(role, account);
    }
```

13.2 setGovernor(newGovernor) X onlyAdmin

```
    function setGovernor(address newGovernor) public onlyAdmin {
        address oldGov = _addressGovernor;
        require(oldGov != newGovernor, "New gov must be different");
        _grantRole(GOVERNOR_ROLE, newGovernor);
        _revokeRole(GOVERNOR_ROLE, oldGov);
        _addressGovernor = newGovernor;
    }
```

13.2 setExecutive(newExecutive) X onlyAdmin

```
    function setExecutive(address newExecutive) public onlyAdmin {
        address oldExec = _addressExecutive;
        require(oldExec != newExecutive, "New exec must be different");
        _grantRole(EXECUTIVE_ROLE, newExecutive);
        _revokeRole(EXECUTIVE_ROLE, oldExec);
        _addressExecutive = newExecutive;
    }
```

13.2 setGuardian(newGuardian) X onlyAdmin

```
    function setGuardian(address newGuardian) public onlyAdmin {
        address oldGuardian = _addressGuardian;
        require(oldGuardian != newGuardian, "New guardian must be different");
        _grantRole(GUARDIAN_ROLE, newGuardian);
        _revokeRole(GUARDIAN_ROLE, oldGuardian);
        _addressGuardian = newGuardian;
    }
```

13.2 _setAndRevokeAnyRole(role, newRoleAddress, oldRoleAddress)

```
    function _setAndRevokeAnyRole(
        bytes32 role,
        address newRoleAddress,
        address oldRoleAddress
    ) internal {
        _grantRole(role, newRoleAddress);
        _revokeRole(role, oldRoleAddress);
    }
```

13.2 getAddressExecutive()

```
    function getAddressExecutive() public view returns (address) {
        return _addressExecutive;
    }
```

13.2 *getAddressGovernor*()

```
    function getAddressGovernor() external view returns (address) {
        return _addressGovernor;
    }
```

13.2 getAddressGuardian()

```
    function getAddressGuardian() external view returns (address) {
        return _addressGuardian;
    }
```

13.2 ***_requireAdmin***()

```
    function _requireAdmin() internal view {
        require(hasRole(ADMIN_ROLE, msg.sender), "Caller is not admin");
    }
```

13.2 ***setAuctioneer***(**newAuctioneer**) X ***onlyAdmin***

```
    function setAuctioneer(address newAuctioneer) public onlyAdmin {
        address oldAuctioneer = _addressAuctioneer;
        require(oldAuctioneer != newAuctioneer, "New Auctioneer must be diff");
        _grantRole(AUCTIONEER, newAuctioneer);
        _revokeRole(AUCTIONEER, oldAuctioneer);
        _addressAuctioneer = newAuctioneer;
    }
```