- 42 Removals + 9 Additions

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;

library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    function mul32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
```

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;

library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    function mul32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
```

```
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }

    function div(uint256 x, uint256 y) internal pure returns(uint256 z){
        require(y > 0);
        z=x/y;
    }
}

library Address {

  function isContract(address account) internal view returns (bool) {
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return _functionCallWithValue(target, data, 0, errorMessage);
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        if (success) {
            return returndata;
        } else {
            if (returndata.length > 0) {
                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

```

```
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }

    function div(uint256 x, uint256 y) internal pure returns(uint256 z){
        require(y > 0);
        z=x/y;
    }
}

library Address {

  function isContract(address account) internal view returns (bool) {
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return _functionCallWithValue(target, data, 0, errorMessage);
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(data);
        if (success) {
            return returndata;
        } else {
            if (returndata.length > 0) {
                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

```

```
    function _verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) private pure returns(bytes memory) {
        if (success) {
            return returndata;
        } else {
            if (returndata.length > 0) {
                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

contract OwnableData {
    address public owner;
    address public pendingOwner;
}

contract Ownable is OwnableData {
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /// @notice `owner` defaults to msg.sender on construction.
    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    /// @notice Transfers ownership to `newOwner`. Either directly or claimable by the new pending owner.
    /// Can only be invoked by the current `owner`.
    /// @param newOwner Address of the new owner.
    /// @param direct True if `newOwner` should be set immediately. False if `newOwner` needs to use `claimOwnership`.
    /// @param renounce Allows the `newOwner` to be `address(0)` if `direct` and `renounce` is True. Has no effect otherwise.
    function transferOwnership(
        address newOwner,
        bool direct,
        bool renounce
    ) public onlyOwner {
        if (direct) {
            // Checks
            require(newOwner != address(0) || renounce, "Ownable: zero address");

            // Effects
            emit OwnershipTransferred(owner, newOwner);
            owner = newOwner;
            pendingOwner = address(0);
        } else {
```

```
            // Effects
            pendingOwner = newOwner;
        }
    }

    /// @notice Needs to be called by `pendingOwner` to claim ownership.
    function claimOwnership() public {
        address _pendingOwner = pendingOwner;

        // Checks
        require(msg.sender == _pendingOwner, "Ownable: caller != pending owner");

        // Effects
        emit OwnershipTransferred(owner, _pendingOwner);
        owner = _pendingOwner;
        pendingOwner = address(0);
    }

    /// @notice Only allows the `owner` to execute the function.
    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: caller is not the owner");
        _;
    }
}

interface IERC20 {
    function decimals() external view returns (uint8);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount) external returns (bool);

    function approve(address spender, uint256 amount) external returns (bool);

    function totalSupply() external view returns (uint256);

    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}

library SafeERC20 {
    using LowGasSafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
```


```
            // Effects
            pendingOwner = newOwner;
        }
    }

    /// @notice Needs to be called by `pendingOwner` to claim ownership.
    function claimOwnership() public {
        address _pendingOwner = pendingOwner;

        // Checks
        require(msg.sender == _pendingOwner, "Ownable: caller != pending owner");

        // Effects
        emit OwnershipTransferred(owner, _pendingOwner);
        owner = _pendingOwner;
        pendingOwner = address(0);
    }

    /// @notice Only allows the `owner` to execute the function.
    modifier onlyOwner() {
        require(msg.sender == owner, "Ownable: caller is not the owner");
        _;
    }
}

interface IERC20 {
    function decimals() external view returns (uint8);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount) external returns (bool);

    function approve(address spender, uint256 amount) external returns (bool);

    function totalSupply() external view returns (uint256);

    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}

library SafeERC20 {
    using LowGasSafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
```

```
    }

    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}

interface IERC20Mintable {
  function mint( uint256 amount_ ) external;

  function mint( address account_, uint256 ammount_ ) external;
}

interface ITIMEERC20 is IERC20Mintable, IERC20 {
    function burnFrom(address account_, uint256 amount_) external;
}

interface IBondCalculator {
  function valuation( address pair_, uint amount_ ) external view returns ( uint _value );
}

contract TimeTreasury is Ownable {

    using LowGasSafeMath for uint;
    using LowGasSafeMath for uint32;
    using SafeERC20 for IERC20;

    event Deposit( address indexed token, uint amount, uint value );
    event Withdrawal( address indexed token, uint amount, uint value );
    event CreateDebt( address indexed debtor, address indexed token, uint amount, uint value );
    event RepayDebt( address indexed debtor, address indexed token, uint amount, uint value );
    event ReservesManaged( address indexed token, uint amount );
    event ReservesUpdated( uint indexed totalReserves );
    event ReservesAudited( uint indexed totalReserves );
    event RewardsMinted( address indexed caller, address indexed recipient, uint amount );
    event ChangeQueued( MANAGING indexed managing, address queued );
    event ChangeActivated( MANAGING indexed managing, address activated, bool result );
    event ChangeLimitAmount( uint256 amount );

    enum MANAGING {
        RESERVEDEPOSITOR,
        RESERVESPENDER,
        RESERVETOKEN,
        RESERVEMANAGER,
        LIQUIDITYDEPOSITOR,
        LIQUIDITYTOKEN,
```


```
    }

    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}

interface IERC20Mintable {
  function mint( uint256 amount_ ) external;

  function mint( address account_, uint256 ammount_ ) external;
}

interface ITIMEERC20 is IERC20Mintable, IERC20 {
    function burnFrom(address account_, uint256 amount_) external;
}

interface IBondCalculator {
  function valuation( address pair_, uint amount_ ) external view returns ( uint _value );
}

contract TimeTreasury is Ownable {

    using LowGasSafeMath for uint;
    using LowGasSafeMath for uint32;
    using SafeERC20 for IERC20;

    event Deposit( address indexed token, uint amount, uint value );
    event Withdrawal( address indexed token, uint amount, uint value );
    event CreateDebt( address indexed debtor, address indexed token, uint amount, uint value );
    event RepayDebt( address indexed debtor, address indexed token, uint amount, uint value );
    event ReservesManaged( address indexed token, uint amount );
    event ReservesUpdated( uint indexed totalReserves );
    event ReservesAudited( uint indexed totalReserves );
    event RewardsMinted( address indexed caller, address indexed recipient, uint amount );
    event ChangeQueued( MANAGING indexed managing, address queued );
    event ChangeActivated( MANAGING indexed managing, address activated, bool result );
    event ChangeLimitAmount( uint256 amount );

    enum MANAGING {
        RESERVEDEPOSITOR,
        RESERVESPENDER,
        RESERVETOKEN,
        RESERVEMANAGER,
        LIQUIDITYDEPOSITOR,
        LIQUIDITYTOKEN,
```

```
        LIQUIDITYMANAGER,
        DEBTOR,
        REWARDMANAGER,
        SOHM
    }

    ITIMEERC20 public immutable Time;
    uint32 public immutable secondsNeededForQueue;

    address[] public reserveTokens; // Push only, beware false-positives.
    mapping( address => bool ) public isReserveToken;
    mapping( address => uint32 ) public reserveTokenQueue; // Delays changes to mapping.

    address[] public reserveDepositors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveDepositor;
    mapping( address => uint32 ) public reserveDepositorQueue; // Delays changes to mapping.

    address[] public reserveSpenders; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveSpender;
    mapping( address => uint32 ) public reserveSpenderQueue; // Delays changes to mapping.

    address[] public liquidityTokens; // Push only, beware false-positives.
    mapping( address => bool ) public isLiquidityToken;
    mapping( address => uint32 ) public LiquidityTokenQueue; // Delays changes to mapping.

    address[] public liquidityDepositors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isLiquidityDepositor;
    mapping( address => uint32 ) public LiquidityDepositorQueue; // Delays changes to mapping.

    mapping( address => address ) public bondCalculator; // bond calculator for liquidity token

    address[] public reserveManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveManager;
    mapping( address => uint32 ) public ReserveManagerQueue; // Delays changes to mapping.

    address[] public liquidityManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isLiquidityManager;
    mapping( address => uint32 ) public LiquidityManagerQueue; // Delays changes to mapping.

    address[] public debtors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isDebtor;
    mapping( address => uint32 ) public debtorQueue; // Delays changes to mapping.
    mapping( address => uint ) public debtorBalance;

```


```
        LIQUIDITYMANAGER,
        DEBTOR,
        REWARDMANAGER,
        SOHM
    }

    ITIMEERC20 public immutable Time;
    uint32 public immutable secondsNeededForQueue;

    address[] public reserveTokens; // Push only, beware false-positives.
    mapping( address => bool ) public isReserveToken;
    mapping( address => uint32 ) public reserveTokenQueue; // Delays changes to mapping.

    address[] public reserveDepositors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveDepositor;
    mapping( address => uint32 ) public reserveDepositorQueue; // Delays changes to mapping.

    address[] public reserveSpenders; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveSpender;
    mapping( address => uint32 ) public reserveSpenderQueue; // Delays changes to mapping.

    address[] public liquidityTokens; // Push only, beware false-positives.
    mapping( address => bool ) public isLiquidityToken;
    mapping( address => uint32 ) public LiquidityTokenQueue; // Delays changes to mapping.

    address[] public liquidityDepositors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isLiquidityDepositor;
    mapping( address => uint32 ) public LiquidityDepositorQueue; // Delays changes to mapping.

    mapping( address => address ) public bondCalculator; // bond calculator for liquidity token

    address[] public reserveManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isReserveManager;
    mapping( address => uint32 ) public ReserveManagerQueue; // Delays changes to mapping.

    address[] public liquidityManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isLiquidityManager;
    mapping( address => uint32 ) public LiquidityManagerQueue; // Delays changes to mapping.

    address[] public debtors; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isDebtor;
    mapping( address => uint32 ) public debtorQueue; // Delays changes to mapping.
    mapping( address => uint ) public debtorBalance;

```

```
    address[] public rewardManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isRewardManager;
    mapping( address => uint32 ) public rewardManagerQueue; // Delays changes to mapping.

    mapping( address => uint256 ) public hourlyLimitAmounts; // tracks amounts
    mapping( address => uint32 ) public hourlyLimitQueue; // Delays changes to mapping.

    uint256 public limitAmount;

    IERC20 public MEMOries;
    uint public sOHMQueue; // Delays change to sOHM address

    uint public totalReserves; // Risk-free value of all assets
    uint public totalDebt;

    constructor (
        address _Time,
        address _MIM,
        uint32 _secondsNeededForQueue,
        uint256 _limitAmount
    ) {
        require( _Time != address(0) );
        Time = ITIMEERC20(_Time);

        isReserveToken[ _MIM ] = true;
        reserveTokens.push( _MIM );

    //    isLiquidityToken[ _OHMDAI ] = true;
    //    liquidityTokens.push( _OHMDAI );

        secondsNeededForQueue = _secondsNeededForQueue;
        limitAmount = _limitAmount;
    }

    function setLimitAmount(uint amount) external onlyOwner {
        limitAmount = amount;
        emit ChangeLimitAmount(limitAmount);
    }

    /**
        @notice allow approved address to deposit an asset for Time
        @param _amount uint
        @param _token address
        @param _profit uint
        @return send_ uint
     */
    function deposit( uint _amount, address _token, uint _profit ) external returns ( uint send_ ) {
        require( isReserveToken[ _token ] || isLiquidityToken[ _token ], "Not accepted" );
        IERC20( _token ).safeTransferFrom( msg.sender, address(this), _amount );

        if ( isReserveToken[ _token ] ) {
            require( isReserveDepositor[ msg.sender ], "Not approved" );
        } else {
            require( isLiquidityDepositor[ msg.sender ], "Not approved" );
        }
```

```
    address[] public rewardManagers; // Push only, beware false-positives. Only for viewing.
    mapping( address => bool ) public isRewardManager;
    mapping( address => uint32 ) public rewardManagerQueue; // Delays changes to mapping.

    IERC20 public MEMOries;
    uint public sOHMQueue; // Delays change to sOHM address

    uint public totalReserves; // Risk-free value of all assets
    uint public totalDebt;

    constructor (
        address _Time,
        address _MIM,
        uint32 _secondsNeededForQueue
    ) {
        require( _Time != address(0) );
        Time = ITIMEERC20(_Time);

        isReserveToken[ _MIM ] = true;
        reserveTokens.push( _MIM );

    //    isLiquidityToken[ _OHMDAI ] = true;
    //    liquidityTokens.push( _OHMDAI );

        secondsNeededForQueue = _secondsNeededForQueue;
    }

    /**
        @notice allow approved address to deposit an asset for Time
        @param _amount uint
        @param _token address
        @param _profit uint
        @return send_ uint
     */
    function deposit( uint _amount, address _token, uint _profit ) external returns ( uint send_ ) {
        require( isReserveToken[ _token ] || isLiquidityToken[ _token ], "Not accepted" );
        IERC20( _token ).safeTransferFrom( msg.sender, address(this), _amount );

        if ( isReserveToken[ _token ] ) {
            require( isReserveDepositor[ msg.sender ], "Not approved" );
        } else {
            require( isLiquidityDepositor[ msg.sender ], "Not approved" );
        }
```

```
365
366         uint value = valueOf(_token, _amount);
367         // mint Time needed and store amount of rewards for distribution
368         send_ = value.sub( _profit );
369         limitRequirements(msg.sender, send_);
370         Time.mint( msg.sender, send_ );
371
372         totalReserves = totalReserves.add( value );
373         emit ReservesUpdated( totalReserves );
374
375         emit Deposit( _token, _amount, value );
376     }
377
378     /**
379         @notice allow approved address to burn Time for reserves
380         @param _amount uint
381         @param _token address
382      */
383     function withdraw( uint _amount, address _token ) external {
384         require( isReserveToken[ _token ], "Not accepted" ); // Only reserves can be used for redemptions
385         require( isReserveSpender[ msg.sender ], "Not approved" );
386
387         uint value = valueOf( _token, _amount );
388         Time.burnFrom( msg.sender, value );
389
390         totalReserves = totalReserves.sub( value );
391         emit ReservesUpdated( totalReserves );
392
393         IERC20( _token ).safeTransfer( msg.sender, _amount );
394
395         emit Withdrawal( _token, _amount, value );
396     }
397
398     /**
399         @notice allow approved address to borrow reserves
400         @param _amount uint
401         @param _token address
402      */
403     function incurDebt( uint _amount, address _token ) external {
404         require( isDebtor[ msg.sender ], "Not approved" );
405         require( isReserveToken[ _token ], "Not accepted" );
406
407         uint value = valueOf( _token, _amount );
408
409         uint maximumDebt = MEMOries.balanceOf( msg.sender ); // Can only borrow against sOHM held
410         uint balance = debtorBalance[ msg.sender ];
411         uint availableDebt = maximumDebt.sub( balance );
412         require( value <= availableDebt, "Exceeds debt limit" );
413         limitRequirements(msg.sender, value);
414         debtorBalance[ msg.sender ] = balance.add( value );
415         totalDebt = totalDebt.add( value );
416
417         totalReserves = totalReserves.sub( value );
```

```
353
354         uint value = valueOfToken(_token, _amount);
355         // mint Time needed and store amount of rewards for distribution
356         send_ = value.sub( _profit );
357         Time.mint( msg.sender, send_ );
358
359         totalReserves = totalReserves.add( value );
360         emit ReservesUpdated( totalReserves );
361
362         emit Deposit( _token, _amount, value );
363     }
364
365     /**
366         @notice allow approved address to burn Time for reserves
367         @param _amount uint
368         @param _token address
369      */
370     function withdraw( uint _amount, address _token ) external {
371         require( isReserveToken[ _token ], "Not accepted" ); // Only reserves can be used for redemptions
372         require( isReserveSpender[ msg.sender ], "Not approved" );
373
374         uint value = valueOfToken( _token, _amount );
375         Time.burnFrom( msg.sender, value );
376
377         totalReserves = totalReserves.sub( value );
378         emit ReservesUpdated( totalReserves );
379
380         IERC20( _token ).safeTransfer( msg.sender, _amount );
381
382         emit Withdrawal( _token, _amount, value );
383     }
384
385     /**
386         @notice allow approved address to borrow reserves
387         @param _amount uint
388         @param _token address
389      */
390     function incurDebt( uint _amount, address _token ) external {
391         require( isDebtor[ msg.sender ], "Not approved" );
392         require( isReserveToken[ _token ], "Not accepted" );
393
394         uint value = valueOfToken( _token, _amount );
395
396         uint maximumDebt = MEMOries.balanceOf( msg.sender ); // Can only borrow against sOHM held
397         uint balance = debtorBalance[ msg.sender ];
398         uint availableDebt = maximumDebt.sub( balance );
399         require( value <= availableDebt, "Exceeds debt limit" );
400         debtorBalance[ msg.sender ] = balance.add( value );
401         totalDebt = totalDebt.add( value );
402
403         totalReserves = totalReserves.sub( value );
```

```
418         emit ReservesUpdated( totalReserves );
419
420         IERC20( _token ).safeTransfer( msg.sender, _amount );
421
422         emit CreateDebt( msg.sender, _token, _amount, value );
423     }
424
425     /**
426         @notice allow approved address to repay borrowed reserves with reserves
427         @param _amount uint
428         @param _token address
429      */
430     function repayDebtWithReserve( uint _amount, address _token ) external {
431         require( isDebtor[ msg.sender ], "Not approved" );
432         require( isReserveToken[ _token ], "Not accepted" );
433
434         IERC20( _token ).safeTransferFrom( msg.sender, address(this), _amount );
435
436         uint value = valueOf( _token, _amount );
437         debtorBalance[ msg.sender ] = debtorBalance[ msg.sender ].sub( value );
438         totalDebt = totalDebt.sub( value );
439
440         totalReserves = totalReserves.add( value );
441         emit ReservesUpdated( totalReserves );
442
443         emit RepayDebt( msg.sender, _token, _amount, value );
444     }
445
446     /**
447         @notice allow approved address to repay borrowed reserves with Time
448         @param _amount uint
449      */
450     function repayDebtWithTime( uint _amount ) external {
451         require( isDebtor[ msg.sender ], "Not approved as debtor" );
452         require( isReserveSpender[ msg.sender ], "Not approved as spender" );
453
454         Time.burnFrom( msg.sender, _amount );
455
456         debtorBalance[ msg.sender ] = debtorBalance[ msg.sender ].sub( _amount );
457         totalDebt = totalDebt.sub( _amount );
458
459         emit RepayDebt( msg.sender, address(Time), _amount, _amount );
460     }
461
462     /**
463         @notice allow approved address to withdraw assets
464         @param _token address
465         @param _amount uint
466      */
467     function manage( address _token, uint _amount ) external {
468         uint value = valueOf(_token, _amount);
469         if( isLiquidityToken[ _token ] ) {
```

```
404         emit ReservesUpdated( totalReserves );
405
406         IERC20( _token ).safeTransfer( msg.sender, _amount );
407
408         emit CreateDebt( msg.sender, _token, _amount, value );
409     }
410
411     /**
412         @notice allow approved address to repay borrowed reserves with reserves
413         @param _amount uint
414         @param _token address
415      */
416     function repayDebtWithReserve( uint _amount, address _token ) external {
417         require( isDebtor[ msg.sender ], "Not approved" );
418         require( isReserveToken[ _token ], "Not accepted" );
419
420         IERC20( _token ).safeTransferFrom( msg.sender, address(this), _amount );
421
422         uint value = valueOfToken( _token, _amount );
423         debtorBalance[ msg.sender ] = debtorBalance[ msg.sender ].sub( value );
424         totalDebt = totalDebt.sub( value );
425
426         totalReserves = totalReserves.add( value );
427         emit ReservesUpdated( totalReserves );
428
429         emit RepayDebt( msg.sender, _token, _amount, value );
430     }
431
432     /**
433         @notice allow approved address to repay borrowed reserves with Time
434         @param _amount uint
435      */
436     function repayDebtWithTime( uint _amount ) external {
437         require( isDebtor[ msg.sender ], "Not approved as debtor" );
438         require( isReserveSpender[ msg.sender ], "Not approved as spender" );
439
440         Time.burnFrom( msg.sender, _amount );
441
442         debtorBalance[ msg.sender ] = debtorBalance[ msg.sender ].sub( _amount );
443         totalDebt = totalDebt.sub( _amount );
444
445         emit RepayDebt( msg.sender, address(Time), _amount, _amount );
446     }
447
448     /**
449         @notice allow approved address to withdraw assets
450         @param _token address
451         @param _amount uint
452      */
453     function manage( address _token, uint _amount ) external {
454         uint value = valueOfToken(_token, _amount);
455         if( isLiquidityToken[ _token ] ) {
```

```
            require( isLiquidityManager[ msg.sender ], "Not approved" );
            require(value <= excessReserves());
        } else {
            if (isReserveToken[ _token ]) require(value <= excessReserves());
            require( isReserveManager[ msg.sender ], "Not approved" );
        }

        limitRequirements(msg.sender, value);
        totalReserves = totalReserves.sub( value );
        emit ReservesUpdated( totalReserves );

        IERC20( _token ).safeTransfer( msg.sender, _amount );

        emit ReservesManaged( _token, _amount );
    }

    /**
        @notice send epoch reward to staking contract
     */
    function mintRewards( address _recipient, uint _amount ) external {
        require( isRewardManager[ msg.sender ], "Not approved" );
        require( _amount <= excessReserves(), "Insufficient reserves" );
        limitRequirements(msg.sender, _amount);
        Time.mint( _recipient, _amount );

        emit RewardsMinted( msg.sender, _recipient, _amount );
    }

    /**
        @notice returns excess reserves not backing tokens
        @return uint
     */
    function excessReserves() public view returns ( uint ) {
        return totalReserves.sub( Time.totalSupply().sub( totalDebt ) );
    }

    /**
        @notice takes inventory of all tracked assets
        @notice always consolidate to recognized reserves before audit
     */
    function auditReserves() external onlyOwner {
        uint reserves;
        for( uint i = 0; i < reserveTokens.length; i++ ) {
            reserves = reserves.add (
                valueOf( reserveTokens[ i ], IERC20( reserveTokens[ i ] ).balanceOf( address(this) ) )
            );
        }
        for( uint i = 0; i < liquidityTokens.length; i++ ) {
            reserves = reserves.add (
```

```
            require( isLiquidityManager[ msg.sender ], "Not approved" );
            require(value <= excessReserves());
        } else {
            if (isReserveToken[ _token ]) require(value <= excessReserves());
            require( isReserveManager[ msg.sender ], "Not approved" );
        }


        totalReserves = totalReserves.sub( value );
        emit ReservesUpdated( totalReserves );

        IERC20( _token ).safeTransfer( msg.sender, _amount );

        emit ReservesManaged( _token, _amount );
    }

    /**
        @notice send epoch reward to staking contract
     */
    function mintRewards( address _recipient, uint _amount ) external {
        require( isRewardManager[ msg.sender ], "Not approved" );
        require( _amount <= excessReserves(), "Insufficient reserves" );

        Time.mint( _recipient, _amount );

        emit RewardsMinted( msg.sender, _recipient, _amount );
    }

    /**
        @notice returns excess reserves not backing tokens
        @return uint
     */
    function excessReserves() public view returns ( uint ) {
        return totalReserves.sub( Time.totalSupply().sub( totalDebt ) );
    }

    /**
        @notice takes inventory of all tracked assets
        @notice always consolidate to recognized reserves before audit
     */
    function auditReserves() external onlyOwner {
        uint reserves;
        for( uint i = 0; i < reserveTokens.length; i++ ) {
            reserves = reserves.add (
                valueOfToken( reserveTokens[ i ], IERC20( reserveTokens[ i ] ).balanceOf( address(this) ) )
            );
        }
        for( uint i = 0; i < liquidityTokens.length; i++ ) {
            reserves = reserves.add (
```

```
                valueOf( liquidityTokens[ i ], IERC20( liquidityTokens[ i ] ).balanceOf( address(this) ) )
            );
        }
        totalReserves = reserves;
        emit ReservesUpdated( reserves );
        emit ReservesAudited( reserves );
    }

    /**
        @notice returns Time valuation of asset
        @param _token address
        @param _amount uint
        @return value_ uint
     */
    function valueOf( address _token, uint _amount ) public view returns ( uint value_ ) {
        if ( isReserveToken[ _token ] ) {
            // convert amount to match Time decimals
            value_ = _amount.mul( 10 ** Time.decimals() ).div( 10 ** IERC20( _token ).decimals() );
        } else if ( isLiquidityToken[ _token ] ) {
            value_ = IBondCalculator( bondCalculator[ _token ] ).valuation( _token, _amount );
        }
    }

    /**
        @notice queue address to change boolean in mapping
        @param _managing MANAGING
        @param _address address
        @return bool
     */
    function queue( MANAGING _managing, address _address ) external onlyOwner returns ( bool ) {
        require( _address != address(0), "IA" );
        if ( _managing == MANAGING.RESERVEDEPOSITOR ) { // 0
            reserveDepositorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVESPENDER ) { // 1
            reserveSpenderQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVETOKEN ) { // 2
            reserveTokenQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVEMANAGER ) { // 3
            ReserveManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue.mul32( 2 ) );
        } else if ( _managing == MANAGING.LIQUIDITYDEPOSITOR ) { // 4
            LiquidityDepositorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.LIQUIDITYTOKEN ) { // 5
            LiquidityTokenQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.LIQUIDITYMANAGER ) { // 6
```

```
                valueOfToken( liquidityTokens[ i ], IERC20( liquidityTokens[ i ] ).balanceOf( address(this) ) )
            );
        }
        totalReserves = reserves;
        emit ReservesUpdated( reserves );
        emit ReservesAudited( reserves );
    }

    /**
        @notice returns Time valuation of asset
        @param _token address
        @param _amount uint
        @return value_ uint
     */
    function valueOfToken( address _token, uint _amount ) public view returns ( uint value_ ) {
        if ( isReserveToken[ _token ] ) {
            // convert amount to match Time decimals
            value_ = _amount.mul( 10 ** Time.decimals() ).div( 10 ** IERC20( _token ).decimals() );
        } else if ( isLiquidityToken[ _token ] ) {
            value_ = IBondCalculator( bondCalculator[ _token ] ).valuation( _token, _amount );
        }
    }

    /**
        @notice queue address to change boolean in mapping
        @param _managing MANAGING
        @param _address address
        @return bool
     */
    function queue( MANAGING _managing, address _address ) external onlyOwner returns ( bool ) {
        require( _address != address(0), "IA" );
        if ( _managing == MANAGING.RESERVEDEPOSITOR ) { // 0
            reserveDepositorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVESPENDER ) { // 1
            reserveSpenderQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVETOKEN ) { // 2
            reserveTokenQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.RESERVEMANAGER ) { // 3
            ReserveManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue.mul32( 2 ) );
        } else if ( _managing == MANAGING.LIQUIDITYDEPOSITOR ) { // 4
            LiquidityDepositorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.LIQUIDITYTOKEN ) { // 5
            LiquidityTokenQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.LIQUIDITYMANAGER ) { // 6
```

```
            LiquidityManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue.mul32( 2 ) );
        } else if ( _managing == MANAGING.DEBTOR ) { // 7
            debtorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.REWARDMANAGER ) { // 8
            rewardManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.SOHM ) { // 9
            sOHMQueue = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else return false;

        emit ChangeQueued( _managing, _address );
        return true;
    }

    /**
        @notice verify queue then set boolean in mapping
        @param _managing MANAGING
        @param _address address
        @param _calculator address
        @return bool
     */
    function toggle(
        MANAGING _managing,
        address _address,
        address _calculator
    ) external onlyOwner returns ( bool ) {
        require( _address != address(0), "IA" );
        bool result;
        if ( _managing == MANAGING.RESERVEDEPOSITOR ) { // 0
            if ( requirements( reserveDepositorQueue, isReserveDepositor, _address ) ) {
                reserveDepositorQueue[ _address ] = 0;
                if( !listContains( reserveDepositors, _address ) ) {
                    reserveDepositors.push( _address );
                }
            }
            result = !isReserveDepositor[ _address ];
            isReserveDepositor[ _address ] = result;

        } else if ( _managing == MANAGING.RESERVESPENDER ) { // 1
            if ( requirements( reserveSpenderQueue, isReserveSpender, _address ) ) {
                reserveSpenderQueue[ _address ] = 0;
                if( !listContains( reserveSpenders, _address ) ) {
                    reserveSpenders.push( _address );
                }
            }
            result = !isReserveSpender[ _address ];
            isReserveSpender[ _address ] = result;
```

```
            LiquidityManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue.mul32( 2 ) );
        } else if ( _managing == MANAGING.DEBTOR ) { // 7
            debtorQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.REWARDMANAGER ) { // 8
            rewardManagerQueue[ _address ] = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else if ( _managing == MANAGING.SOHM ) { // 9
            sOHMQueue = uint32(block.timestamp).add32( secondsNeededForQueue );
        } else return false;

        emit ChangeQueued( _managing, _address );
        return true;
    }

    /**
        @notice verify queue then set boolean in mapping
        @param _managing MANAGING
        @param _address address
        @param _calculator address
        @return bool
     */
    function toggle(
        MANAGING _managing,
        address _address,
        address _calculator
    ) external onlyOwner returns ( bool ) {
        require( _address != address(0), "IA" );
        bool result;
        if ( _managing == MANAGING.RESERVEDEPOSITOR ) { // 0
            if ( requirements( reserveDepositorQueue, isReserveDepositor, _address ) ) {
                reserveDepositorQueue[ _address ] = 0;
                if( !listContains( reserveDepositors, _address ) ) {
                    reserveDepositors.push( _address );
                }
            }
            result = !isReserveDepositor[ _address ];
            isReserveDepositor[ _address ] = result;

        } else if ( _managing == MANAGING.RESERVESPENDER ) { // 1
            if ( requirements( reserveSpenderQueue, isReserveSpender, _address ) ) {
                reserveSpenderQueue[ _address ] = 0;
                if( !listContains( reserveSpenders, _address ) ) {
                    reserveSpenders.push( _address );
                }
            }
            result = !isReserveSpender[ _address ];
            isReserveSpender[ _address ] = result;
```

```
        } else if ( _managing == MANAGING.RESERVETOKEN ) { // 2
            if ( requirements( reserveTokenQueue, isReserveToken, _address ) ) {
                reserveTokenQueue[ _address ] = 0;
                if( !listContains( reserveTokens, _address ) && !listContains( liquidityTokens, _address ) ) {
                    reserveTokens.push( _address );
                }
            }
            result = !isReserveToken[ _address ];
            require(!result || !isLiquidityToken[_address], "Do not add to both types of token");
            isReserveToken[ _address ] = result;

        } else if ( _managing == MANAGING.RESERVEMANAGER ) { // 3
            if ( requirements( ReserveManagerQueue, isReserveManager, _address ) ) {
                reserveManagers.push( _address );
                ReserveManagerQueue[ _address ] = 0;
                if( !listContains( reserveManagers, _address ) ) {
                    reserveManagers.push( _address );
                }
            }
            result = !isReserveManager[ _address ];
            isReserveManager[ _address ] = result;

        } else if ( _managing == MANAGING.LIQUIDITYDEPOSITOR ) { // 4
            if ( requirements( LiquidityDepositorQueue, isLiquidityDepositor, _address ) ) {
                liquidityDepositors.push( _address );
                LiquidityDepositorQueue[ _address ] = 0;
                if( !listContains( liquidityDepositors, _address ) ) {
                    liquidityDepositors.push( _address );
                }
            }
            result = !isLiquidityDepositor[ _address ];
            isLiquidityDepositor[ _address ] = result;

        } else if ( _managing == MANAGING.LIQUIDITYTOKEN ) { // 5
            if ( requirements( LiquidityTokenQueue, isLiquidityToken, _address ) ) {
                LiquidityTokenQueue[ _address ] = 0;
                if( !listContains( liquidityTokens, _address ) && !listContains( reserveTokens, _address ) ) {
                    liquidityTokens.push( _address );
                }
            }
            result = !isLiquidityToken[ _address ];
            require(!result || !isReserveToken[_address], "Do not add to both types of token");
            isLiquidityToken[ _address ] = result;
```

```
        } else if ( _managing == MANAGING.RESERVETOKEN ) { // 2
            if ( requirements( reserveTokenQueue, isReserveToken, _address ) ) {
                reserveTokenQueue[ _address ] = 0;
                if( !listContains( reserveTokens, _address ) && !listContains( liquidityTokens, _address ) ) {
                    reserveTokens.push( _address );
                }
            }
            result = !isReserveToken[ _address ];
            require(!result || !isLiquidityToken[_address], "Do not add to both types of token");
            isReserveToken[ _address ] = result;

        } else if ( _managing == MANAGING.RESERVEMANAGER ) { // 3
            if ( requirements( ReserveManagerQueue, isReserveManager, _address ) ) {
                reserveManagers.push( _address );
                ReserveManagerQueue[ _address ] = 0;
                if( !listContains( reserveManagers, _address ) ) {
                    reserveManagers.push( _address );
                }
            }
            result = !isReserveManager[ _address ];
            isReserveManager[ _address ] = result;

        } else if ( _managing == MANAGING.LIQUIDITYDEPOSITOR ) { // 4
            if ( requirements( LiquidityDepositorQueue, isLiquidityDepositor, _address ) ) {
                liquidityDepositors.push( _address );
                LiquidityDepositorQueue[ _address ] = 0;
                if( !listContains( liquidityDepositors, _address ) ) {
                    liquidityDepositors.push( _address );
                }
            }
            result = !isLiquidityDepositor[ _address ];
            isLiquidityDepositor[ _address ] = result;

        } else if ( _managing == MANAGING.LIQUIDITYTOKEN ) { // 5
            if ( requirements( LiquidityTokenQueue, isLiquidityToken, _address ) ) {
                LiquidityTokenQueue[ _address ] = 0;
                if( !listContains( liquidityTokens, _address ) && !listContains( reserveTokens, _address ) ) {
                    liquidityTokens.push( _address );
                }
            }
            result = !isLiquidityToken[ _address ];
            require(!result || !isReserveToken[_address], "Do not add to both types of token");
            isLiquidityToken[ _address ] = result;
```

```
            bondCalculator[ _address ] = _calculator;

        } else if ( _managing == MANAGING.LIQUIDITY MANAGER ) { // 6
            if ( requirements( LiquidityManagerQueue, isLiquidityManager, _address ) ) {
                LiquidityManagerQueue[ _address ] = 0;
                if( !listContains( liquidityManagers, _address ) ) {
                    liquidityManagers.push( _address );
                }
            }
            result = !isLiquidityManager[ _address ];
            isLiquidityManager[ _address ] = result;

        } else if ( _managing == MANAGING.DEBTOR ) { // 7
            if ( requirements( debtorQueue, isDebtor, _address ) ) {
                debtorQueue[ _address ] = 0;
                if( !listContains( debtors, _address ) ) {
                    debtors.push( _address );
                }
            }
            result = !isDebtor[ _address ];
            isDebtor[ _address ] = result;

        } else if ( _managing == MANAGING.REWARDMANAGER ) { // 8
            if ( requirements( rewardManagerQueue, isRewardManager, _address ) ) {
                rewardManagerQueue[ _address ] = 0;
                if( !listContains( rewardManagers, _address ) ) {
                    rewardManagers.push( _address );
                }
            }
            result = !isRewardManager[ _address ];
            isRewardManager[ _address ] = result;

        } else if ( _managing == MANAGING.SOHM ) { // 9
            sOHMQueue = 0;
            MEMOries = IERC20(_address);
            result = true;

        } else return false;

        emit ChangeActivated( _managing, _address, result );
        return true;
    }

    /**
        @notice checks requirements and returns altered structs
        @param queue_ mapping( address => uint )
        @param status_ mapping( address => bool )
        @param _address address
        @return bool
     */
    function requirements(
```

```
            bondCalculator[ _address ] = _calculator;

        } else if ( _managing == MANAGING.LIQUIDITY MANAGER ) { // 6
            if ( requirements( LiquidityManagerQueue, isLiquidityManager, _address ) ) {
                LiquidityManagerQueue[ _address ] = 0;
                if( !listContains( liquidityManagers, _address ) ) {
                    liquidityManagers.push( _address );
                }
            }
            result = !isLiquidityManager[ _address ];
            isLiquidityManager[ _address ] = result;

        } else if ( _managing == MANAGING.DEBTOR ) { // 7
            if ( requirements( debtorQueue, isDebtor, _address ) ) {
                debtorQueue[ _address ] = 0;
                if( !listContains( debtors, _address ) ) {
                    debtors.push( _address );
                }
            }
            result = !isDebtor[ _address ];
            isDebtor[ _address ] = result;

        } else if ( _managing == MANAGING.REWARDMANAGER ) { // 8
            if ( requirements( rewardManagerQueue, isRewardManager, _address ) ) {
                rewardManagerQueue[ _address ] = 0;
                if( !listContains( rewardManagers, _address ) ) {
                    rewardManagers.push( _address );
                }
            }
            result = !isRewardManager[ _address ];
            isRewardManager[ _address ] = result;

        } else if ( _managing == MANAGING.SOHM ) { // 9
            sOHMQueue = 0;
            MEMOries = IERC20(_address);
            result = true;

        } else return false;

        emit ChangeActivated( _managing, _address, result );
        return true;
    }

    /**
        @notice checks requirements and returns altered structs
        @param queue_ mapping( address => uint )
        @param status_ mapping( address => bool )
        @param _address address
        @return bool
     */
    function requirements(
```

```
        mapping( address => uint32 ) storage queue_,
        mapping( address => bool ) storage status_,
        address _address
    ) internal view returns ( bool ) {
        if ( !status_[ _address ] ) {
            require( queue_[ _address ] != 0, "Must queue" );
            require( queue_[ _address ] <= uint32(block.timestamp), "Queue not expired" );
            return true;
        } return false;
    }

    /**
        @notice checks LimitRequirements
        @param _address address
        @param _address value
     */
    function limitRequirements(
        address _address,
        uint256 value
    ) internal {
        if (block.timestamp.sub(hourlyLimitQueue[_address]) >= 1 hours)
        {
            hourlyLimitAmounts[_address] = limitAmount;
            hourlyLimitQueue[_address] = uint32(block.timestamp);
        }
        hourlyLimitAmounts[_address] = hourlyLimitAmounts[_address].sub(value);
    }

    /**
        @notice checks array to ensure against duplicate
        @param _list address[]
        @param _token address
        @return bool
     */
    function listContains( address[] storage _list, address _token ) internal view returns ( bool ) {
        for( uint i = 0; i < _list.length; i++ ) {
            if( _list[ i ] == _token ) {
                return true;
            }
        }
        return false;
    }
}
```

```
        mapping( address => uint32 ) storage queue_,
        mapping( address => bool ) storage status_,
        address _address
    ) internal view returns ( bool ) {
        if ( !status_[ _address ] ) {
            require( queue_[ _address ] != 0, "Must queue" );
            require( queue_[ _address ] <= uint32(block.timestamp), "Queue not expired" );
            return true;
        } return false;
    }

    /**
        @notice checks array to ensure against duplicate
        @param _list address[]
        @param _token address
        @return bool
     */
    function listContains( address[] storage _list, address _token ) internal view returns ( bool ) {
        for( uint i = 0; i < _list.length; i++ ) {
            if( _list[ i ] == _token ) {
                return true;
            }
        }
        return false;
    }
}
```