- 13 Removals + 29 Additions

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;
pragma abicoder v2;

interface IOwnable {
  function policy() external view returns (address);

  function renounceManagement() external;

  function pushManagement( address newOwner_ ) external;

  function pullManagement() external;
}

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;
        }
```

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;
pragma abicoder v2;

interface IOwnable {
  function policy() external view returns (address);

  function renounceManagement() external;

  function pushManagement( address newOwner_ ) external;

  function pullManagement() external;
}

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");
        _;
    }
}

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);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
```

```
    }

    /// @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");
        _;
    }
}

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);
    }

    /// @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));
    }
}

library Address {

    function isContract(address account) internal view returns (bool) {

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

    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    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 value) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
```

```
    /// @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) {

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

    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    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 value) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
    }

    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, 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 {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
```

```
    }

    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, 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 {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, 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) {

                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

    function addressToString(address _address) internal pure returns(string memory) {
        bytes32 _bytes = bytes32(uint256(_address));
        bytes memory HEX = "0123456789abcdef";
        bytes memory _addr = new bytes(42);

        _addr[0] = '0';
        _addr[1] = 'x';

        for(uint256 i = 0; i < 20; i++) {
            _addr[2+i*2] = HEX[uint8(_bytes[i + 12] >> 4)];
```

```
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, 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) {

                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }

    function addressToString(address _address) internal pure returns(string memory) {
        bytes32 _bytes = bytes32(uint256(_address));
        bytes memory HEX = "0123456789abcdef";
        bytes memory _addr = new bytes(42);

        _addr[0] = '0';
        _addr[1] = 'x';

        for(uint256 i = 0; i < 20; i++) {
            _addr[2+i*2] = HEX[uint8(_bytes[i + 12] >> 4)];
            _addr[3+i*2] = HEX[uint8(_bytes[i + 12] & 0x0f)];
        }

        return string(_addr);

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

    function totalSupply() external view returns (uint256);

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

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

    function allowance(address owner, address spender) external view returns (uint256);

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

    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 safeApprove(IERC20 token, address spender, uint256 value) internal {

        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
```

```
            _addr[3+i*2] = HEX[uint8(_bytes[i + 12] & 0x0f)];
        }

        return string(_addr);

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

    function totalSupply() external view returns (uint256);

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

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

    function allowance(address owner, address spender) external view returns (uint256);

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

    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 safeApprove(IERC20 token, address spender, uint256 value) internal {

        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender)
            .sub(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    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");
        }
    }
}

library FullMath {
    function fullMul(uint256 x, uint256 y) private pure returns (uint256 l, uint256 h) {
        uint256 mm = mulmod(x, y, uint256(-1));
        l = x * y;
        h = mm - l;
        if (mm < l) h -= 1;
    }

    function fullDiv(
        uint256 l,
        uint256 h,
        uint256 d
    ) private pure returns (uint256) {
        uint256 pow2 = d & -d;
        d /= pow2;
        l /= pow2;
        l += h * ((-pow2) / pow2 + 1);
        uint256 r = 1;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        return l * r;
    }

    function mulDiv(
        uint256 x,
        uint256 y,
        uint256 d
    ) internal pure returns (uint256) {
        (uint256 l, uint256 h) = fullMul(x, y);
        uint256 mm = mulmod(x, y, d);
        if (mm > l) h -= 1;
```

```
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender)
            .sub(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    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");
        }
    }
}

library FullMath {
    function fullMul(uint256 x, uint256 y) private pure returns (uint256 l, uint256 h) {
        uint256 mm = mulmod(x, y, uint256(-1));
        l = x * y;
        h = mm - l;
        if (mm < l) h -= 1;
    }

    function fullDiv(
        uint256 l,
        uint256 h,
        uint256 d
    ) private pure returns (uint256) {
        uint256 pow2 = d & -d;
        d /= pow2;
        l /= pow2;
        l += h * ((-pow2) / pow2 + 1);
        uint256 r = 1;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        return l * r;
    }

    function mulDiv(
        uint256 x,
        uint256 y,
        uint256 d
    ) internal pure returns (uint256) {
        (uint256 l, uint256 h) = fullMul(x, y);
        uint256 mm = mulmod(x, y, d);
        if (mm > l) h -= 1;
        l -= mm;
        require(h < d, 'FullMath::mulDiv: overflow');
        return fullDiv(l, h, d);
    }
}

library FixedPoint {

    struct uq112x112 {
        uint224 _x;
    }

    struct uq144x112 {
        uint256 _x;
    }

    uint8 private constant RESOLUTION = 112;
    uint256 private constant Q112 = 0x10000000000000000000000000000;
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000000000000;
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // decimal of UQ*x112 (lower 112 bits)

    function decode(uq112x112 memory self) internal pure returns (uint112) {
        return uint112(self._x >> RESOLUTION);
    }

    function decode112with18(uq112x112 memory self) internal pure returns (uint) {

        return uint(self._x) / 5192296858534827;
    }

    function fraction(uint256 numerator, uint256 denominator) internal pure returns (uq112x112 memory) {
        require(denominator > 0, 'FixedPoint::fraction: division by zero');
        if (numerator == 0) return FixedPoint.uq112x112(0);

        if (numerator <= uint144(-1)) {
            uint256 result = (numerator << RESOLUTION) / denominator;
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        } else {
            uint256 result = FullMath.mulDiv(numerator, Q112, denominator);
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        }
    }
}

interface ITreasury {
    function deposit( uint _amount, address _token, uint _profit ) external returns ( bool );
    function valueOf( address _token, uint _amount ) external view returns ( uint value_ );
}

interface IBondCalculator {
```

```
        l -= mm;
        require(h < d, 'FullMath::mulDiv: overflow');
        return fullDiv(l, h, d);
    }
}

library FixedPoint {

    struct uq112x112 {
        uint224 _x;
    }

    struct uq144x112 {
        uint256 _x;
    }

    uint8 private constant RESOLUTION = 112;
    uint256 private constant Q112 = 0x10000000000000000000000000000;
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000000000000;
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // decimal of UQ*x112 (lower 112 bits)

    function decode(uq112x112 memory self) internal pure returns (uint112) {
        return uint112(self._x >> RESOLUTION);
    }

    function decode112with18(uq112x112 memory self) internal pure returns (uint) {

        return uint(self._x) / 5192296858534827;
    }

    function fraction(uint256 numerator, uint256 denominator) internal pure returns (uq112x112 memory) {
        require(denominator > 0, 'FixedPoint::fraction: division by zero');
        if (numerator == 0) return FixedPoint.uq112x112(0);

        if (numerator <= uint144(-1)) {
            uint256 result = (numerator << RESOLUTION) / denominator;
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        } else {
            uint256 result = FullMath.mulDiv(numerator, Q112, denominator);
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        }
    }
}

interface ITreasury {
    function deposit( uint _amount, address _token, uint _profit ) external returns ( uint );
    function valueOfToken( address _token, uint _amount ) external view returns ( uint value_ );
}

interface IBondCalculator {
    function valuation( address _LP, uint _amount ) external view returns ( uint );
    function markdown( address _LP ) external view returns ( uint );
}

interface IStaking {
    function stake( uint _amount, address _recipient ) external returns ( bool );
}

interface IStakingHelper {
    function stake( uint _amount, address _recipient ) external;
}

contract TimeBondDepository is Ownable {

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




    /* ======== EVENTS ======== */

    event BondCreated( uint deposit, uint indexed payout, uint indexed expires, uint indexed priceInUSD );
    event BondRedeemed( address indexed recipient, uint payout, uint remaining );
    event BondPriceChanged( uint indexed priceInUSD, uint indexed internalPrice, uint indexed debtRatio );
    event ControlVariableAdjustment( uint initialBCV, uint newBCV, uint adjustment, bool addition );
    event InitTerms( Terms terms);
    event LogSetTerms(PARAMETER param, uint value);
    event LogSetAdjustment( Adjust adjust);
    event LogSetStaking( address indexed stakingContract, bool isHelper);
    event LogRecoverLostToken( address indexed tokenToRecover, uint amount);



    /* ======== STATE VARIABLES ======== */

    IERC20 public immutable Time; // token given as payment for bond
    IERC20 public immutable principle; // token used to create bond
    ITreasury public immutable treasury; // mints Time when receives principle
    address public immutable DAO; // receives profit share from bond

    bool public immutable isLiquidityBond; // LP and Reserve bonds are treated slightly different
    IBondCalculator public immutable bondCalculator; // calculates value of LP tokens

    IStaking public staking; // to auto-stake payout
    IStakingHelper public stakingHelper; // to stake and claim if no staking warmup
    bool public useHelper;

```

```
    function valuation( address _LP, uint _amount ) external view returns ( uint );
    function markdown( address _LP ) external view returns ( uint );
}

interface IStaking {
    function stake( uint _amount, address _recipient ) external returns ( bool );
}

interface IStakingHelper {
    function stake( uint _amount, address _recipient ) external;
}

contract MaiaBondDepository is Ownable {

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




    /* ======== EVENTS ======== */

    event BondCreated( uint deposit, uint indexed payout, uint indexed expires, uint indexed priceInUSD );
    event BondRedeemed( address indexed recipient, uint payout, uint remaining );
    event BondPriceChanged( uint indexed priceInUSD, uint indexed internalPrice, uint indexed debtRatio );
    event ControlVariableAdjustment( uint initialBCV, uint newBCV, uint adjustment, bool addition );
    event InitTerms( Terms terms);
    event LogSetTerms(PARAMETER param, uint value);
    event LogSetAdjustment( Adjust adjust);
    event LogSetStaking( address indexed stakingContract, bool isHelper);
    event LogRecoverLostToken( address indexed tokenToRecover, uint amount);



    /* ======== STATE VARIABLES ======== */

    IERC20 public immutable Time; // token given as payment for bond
    IERC20 public immutable principle; // token used to create bond
    ITreasury public immutable treasury; // mints Time when receives principle
    address public immutable DAO; // receives profit share from bond

    bool public immutable isLiquidityBond; // LP and Reserve bonds are treated slightly different
    IBondCalculator public immutable bondCalculator; // calculates value of LP tokens

    IStaking public staking; // to auto-stake payout
    IStakingHelper public stakingHelper; // to stake and claim if no staking warmup
    bool public useHelper;

    Terms public terms; // stores terms for new bonds
    Adjust public adjustment; // stores adjustment to BCV data

    mapping( address => Bond ) public bondInfo; // stores bond information for depositors

    uint public totalDebt; // total value of outstanding bonds; used for pricing
    uint32 public lastDecay; // reference time for debt decay

    mapping (address => bool) public allowedZappers;




    /* ======== STRUCTS ======== */

    // Info for creating new bonds
    struct Terms {
        uint controlVariable; // scaling variable for price
        uint minimumPrice; // vs principle value
        uint maxPayout; // in thousandths of a %. i.e. 500 = 0.5%
        uint fee; // as % of bond payout, in hundreths. ( 500 = 5% = 0.05 for every 1 paid)
        uint maxDebt; // 9 decimal debt ratio, max % total supply created as debt
        uint32 vestingTerm; // in seconds
    }

    // Info for bond holder
    struct Bond {
        uint payout; // Time remaining to be paid
        uint pricePaid; // In DAI, for front end viewing
        uint32 lastTime; // Last interaction
        uint32 vesting; // Seconds left to vest
    }

    // Info for incremental adjustments to control variable
    struct Adjust {
        bool add; // addition or subtraction
        uint rate; // increment
        uint target; // BCV when adjustment finished
        uint32 buffer; // minimum length (in seconds) between adjustments
        uint32 lastTime; // time when last adjustment made
    }




    /* ======== INITIALIZATION ======== */

    constructor (
        address _Time,
        address _principle,
        address _treasury,
        address _DAO,
        address _bondCalculator
    ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
```

```
    Terms public terms; // stores terms for new bonds
    Adjust public adjustment; // stores adjustment to BCV data

    mapping( address => Bond ) public bondInfo; // stores bond information for depositors

    uint public totalDebt; // total value of outstanding bonds; used for pricing
    uint32 public lastDecay; // reference time for debt decay

    mapping (address => bool) public allowedZappers;




    /* ======== STRUCTS ======== */

    // Info for creating new bonds
    struct Terms {
        uint controlVariable; // scaling variable for price
        uint minimumPrice; // vs principle value
        uint maxPayout; // in thousandths of a %. i.e. 500 = 0.5%
        uint fee; // as % of bond payout, in hundreths. ( 500 = 5% = 0.05 for every 1 paid)
        uint maxDebt; // 9 decimal debt ratio, max % total supply created as debt
        uint32 vestingTerm; // in seconds
    }

    // Info for bond holder
    struct Bond {
        uint payout; // Time remaining to be paid
        uint pricePaid; // In DAI, for front end viewing
        uint32 lastTime; // Last interaction
        uint32 vesting; // Seconds left to vest
    }

    // Info for incremental adjustments to control variable
    struct Adjust {
        bool add; // addition or subtraction
        uint rate; // increment
        uint target; // BCV when adjustment finished
        uint32 buffer; // minimum length (in seconds) between adjustments
        uint32 lastTime; // time when last adjustment made
    }




    /* ======== INITIALIZATION ======== */

    constructor (
        address _Time,
        address _principle,
        address _treasury,
        address _DAO,
        address _bondCalculator
    ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
        require( _principle != address(0) );
        principle = IERC20(_principle);
        require( _treasury != address(0) );
        treasury = ITreasury(_treasury);
        require( _DAO != address(0) );
        DAO = _DAO;
        // bondCalculator should be address(0) if not LP bond
        bondCalculator = IBondCalculator(_bondCalculator);
        isLiquidityBond = ( _bondCalculator != address(0) );
    }

    /**
     *  @notice initializes bond parameters
     *  @param _controlVariable uint
     *  @param _vestingTerm uint32
     *  @param _minimumPrice uint
     *  @param _maxPayout uint
     *  @param _fee uint
     *  @param _maxDebt uint
     */
    function initializeBondTerms(
        uint _controlVariable,
        uint _minimumPrice,
        uint _maxPayout,
        uint _fee,
        uint _maxDebt,
        uint32 _vestingTerm
    ) external onlyOwner() {
        require( terms.controlVariable == 0, "Bonds must be initialized from 0" );
        require( _controlVariable >= 40, "Can lock adjustment" );
        require( _maxPayout <= 1000, "Payout cannot be above 1 percent" );
        require( _vestingTerm >= 129600, "Vesting must be longer than 36 hours" );
        require( _fee <= 10000, "DAO fee cannot exceed payout" );
        terms = Terms ({
            controlVariable: _controlVariable,
            minimumPrice: _minimumPrice,
            maxPayout: _maxPayout,
            fee: _fee,
            maxDebt: _maxDebt,
            vestingTerm: _vestingTerm
        });
        lastDecay = uint32(block.timestamp);
        emit InitTerms(terms);
    }




    /* ======== POLICY FUNCTIONS ======== */

    enum PARAMETER { VESTING, PAYOUT, FEE, DEBT, MINPRICE }
    /**
     *  @notice set parameters for new bonds
     *  @param _parameter PARAMETER
     *  @param _input uint
     */
    function setBondTerms ( PARAMETER _parameter, uint _input ) external onlyOwner() {
```

```
        require( _principle != address(0) );
        principle = IERC20(_principle);
        require( _treasury != address(0) );
        treasury = ITreasury(_treasury);
        require( _DAO != address(0) );
        DAO = _DAO;
        // bondCalculator should be address(0) if not LP bond
        bondCalculator = IBondCalculator(_bondCalculator);
        isLiquidityBond = ( _bondCalculator != address(0) );
    }

    /**
     *  @notice initializes bond parameters
     *  @param _controlVariable uint
     *  @param _vestingTerm uint32
     *  @param _minimumPrice uint
     *  @param _maxPayout uint
     *  @param _fee uint
     *  @param _maxDebt uint
     */
    function initializeBondTerms(
        uint _controlVariable,
        uint _minimumPrice,
        uint _maxPayout,
        uint _fee,
        uint _maxDebt,
        uint32 _vestingTerm
    ) external onlyOwner() {
        require( terms.controlVariable == 0, "Bonds must be initialized from 0" );
        require( _controlVariable >= 40, "Can lock adjustment" );
        require( _maxPayout <= 2000, "Payout cannot be above 1 percent" );
        require( _vestingTerm >= 129600, "Vesting must be longer than 36 hours" );
        require( _fee <= 10000, "DAO fee cannot exceed payout" );
        terms = Terms ({
            controlVariable: _controlVariable,
            minimumPrice: _minimumPrice,
            maxPayout: _maxPayout,
            fee: _fee,
            maxDebt: _maxDebt,
            vestingTerm: _vestingTerm
        });
        lastDecay = uint32(block.timestamp);
        emit InitTerms(terms);
    }




    /* ======== POLICY FUNCTIONS ======== */

    enum PARAMETER { VESTING, PAYOUT, FEE, DEBT, MINPRICE }
    /**
     *  @notice set parameters for new bonds
     *  @param _parameter PARAMETER
     *  @param _input uint
     */
    function setBondTerms ( PARAMETER _parameter, uint _input ) external onlyOwner() {
        if ( _parameter == PARAMETER.VESTING ) { // 0
            require( _input >= 129600, "Vesting must be longer than 36 hours" );
            terms.vestingTerm = uint32(_input);
        } else if ( _parameter == PARAMETER.PAYOUT ) { // 1
            require( _input <= 1000, "Payout cannot be above 1 percent" );
            terms.maxPayout = _input;
        } else if ( _parameter == PARAMETER.FEE ) { // 2
            require( _input <= 10000, "DAO fee cannot exceed payout" );
            terms.fee = _input;
        } else if ( _parameter == PARAMETER.DEBT ) { // 3
            terms.maxDebt = _input;
        } else if ( _parameter == PARAMETER.MINPRICE ) { // 4
            terms.minimumPrice = _input;
        }
        emit LogSetTerms(_parameter, _input);
    }

    /**
     *  @notice set control variable adjustment
     *  @param _addition bool
     *  @param _increment uint
     *  @param _target uint
     *  @param _buffer uint
     */
    function setAdjustment (
        bool _addition,
        uint _increment,
        uint _target,
        uint32 _buffer
    ) external onlyOwner() {
        require( _increment <= terms.controlVariable.mul( 25 ) / 1000 , "Increment too large" );
        require(_target >= 40, "Next Adjustment could be locked");
        adjustment = Adjust({
            add: _addition,
            rate: _increment,
            target: _target,
            buffer: _buffer,
            lastTime: uint32(block.timestamp)
        });
        emit LogSetAdjustment(adjustment);
    }

    /**
     *  @notice set contract for auto stake
     *  @param _staking address
     *  @param _helper bool
     */
    function setStaking( address _staking, bool _helper ) external onlyOwner() {
        require( _staking != address(0), "IA" );
        if ( _helper ) {
            useHelper = true;
            stakingHelper = IStakingHelper(_staking);
        } else {
            useHelper = false;
            staking = IStaking(_staking);
        }
```

```
        if ( _parameter == PARAMETER.VESTING ) { // 0
            require( _input >= 129600, "Vesting must be longer than 36 hours" );
            terms.vestingTerm = uint32(_input);
        } else if ( _parameter == PARAMETER.PAYOUT ) { // 1
            require( _input <= 2000, "Payout cannot be above 1 percent" );
            terms.maxPayout = _input;
        } else if ( _parameter == PARAMETER.FEE ) { // 2
            require( _input <= 10000, "DAO fee cannot exceed payout" );
            terms.fee = _input;
        } else if ( _parameter == PARAMETER.DEBT ) { // 3
            terms.maxDebt = _input;
        } else if ( _parameter == PARAMETER.MINPRICE ) { // 4
            terms.minimumPrice = _input;
        }
        emit LogSetTerms(_parameter, _input);
    }

    /**
     *  @notice set control variable adjustment
     *  @param _addition bool
     *  @param _increment uint
     *  @param _target uint
     *  @param _buffer uint
     */
    function setAdjustment (
        bool _addition,
        uint _increment,
        uint _target,
        uint32 _buffer
    ) external onlyOwner() {
        require( _increment <= terms.controlVariable.mul( 25 ) / 1000 , "Increment too large" );
        require(_target >= 40, "Next Adjustment could be locked");
        adjustment = Adjust({
            add: _addition,
            rate: _increment,
            target: _target,
            buffer: _buffer,
            lastTime: uint32(block.timestamp)
        });
        emit LogSetAdjustment(adjustment);
    }

    /**
     *  @notice set contract for auto stake
     *  @param _staking address
     *  @param _helper bool
     */
    function setStaking( address _staking, bool _helper ) external onlyOwner() {
        require( _staking != address(0), "IA" );
        if ( _helper ) {
            useHelper = true;
            stakingHelper = IStakingHelper(_staking);
        } else {
            useHelper = false;
            staking = IStaking(_staking);
        }
        emit LogSetStaking(_staking, _helper);
    }

    function allowZapper(address zapper) external onlyOwner {
        require(zapper != address(0), "ZNA");

        allowedZappers[zapper] = true;
    }

    function removeZapper(address zapper) external onlyOwner {

        allowedZappers[zapper] = false;
    }




    /* ======== USER FUNCTIONS ======== */

    /**
     *  @notice deposit bond
     *  @param _amount uint
     *  @param _maxPrice uint
     *  @param _depositor address
     *  @return uint
     */
    function deposit(
        uint _amount,
        uint _maxPrice,
        address _depositor
    ) external returns ( uint ) {
        require( _depositor != address(0), "Invalid address" );
        require(msg.sender == _depositor || allowedZappers[msg.sender], "LFNA");
        decayDebt();


        uint priceInUSD = bondPriceInUSD(); // Stored in bond info
        uint nativePrice = _bondPrice();

        require( _maxPrice >= nativePrice, "Slippage limit: more than max price" ); // slippage protection

        uint value = treasury.valueOf( address(principle), _amount );
        uint payout = payoutFor( value ); // payout to bonder is computed
        require( totalDebt.add(value) <= terms.maxDebt, "Max capacity reached" );
        require( payout >= 10000000, "Bond too small" ); // must be > 0.01 Time ( underflow protection )
        require( payout <= maxPayout(), "Bond too large"); // size protection because there is no slippage

        // profits are calculated
        uint fee = payout.mul( terms.fee )/ 10000 ;
        uint profit = value.sub( payout ).sub( fee );

        uint balanceBefore = Time.balanceOf(address(this));
        /**
```

```
        emit LogSetStaking(_staking, _helper);
    }

    function allowZapper(address zapper) external onlyOwner {
        require(zapper != address(0), "ZNA");

        allowedZappers[zapper] = true;
    }

    function removeZapper(address zapper) external onlyOwner {

        allowedZappers[zapper] = false;
    }




    /* ======== USER FUNCTIONS ======== */

    /**
     *  @notice deposit bond
     *  @param _amount uint
     *  @param _maxPrice uint
     *  @param _depositor address
     *  @return uint
     */
    function deposit(
        uint _amount,
        uint _maxPrice,
        address _depositor
    ) external returns ( uint ) {
        require( _depositor != address(0), "Invalid address" );
        require(msg.sender == _depositor || allowedZappers[msg.sender], "LFNA");
        decayDebt();
        uint priceInUSD = bondPriceInUSD(); // Stored in bond info
        uint nativePrice = _bondPrice();

        require( _maxPrice >= nativePrice, "Slippage limit: more than max price" ); // slippage protection

        uint value = treasury.valueOfToken( address(principle), _amount );
        uint payout = payoutFor( value ); // payout to bonder is computed
        require( totalDebt.add(value) <= terms.maxDebt, "Max capacity reached" );
        require( payout >= 10000000, "Bond too small" ); // must be > 0.01 Time ( underflow protection )
        require( payout <= maxPayout(), "Bond too large"); // size protection because there is no slippage

        // profits are calculated
        uint fee = (payout.mul( terms.fee )).div(10000);
        uint profit = value.sub( payout ).sub( fee );

        uint balanceBefore = Time.balanceOf(address(this));
        /**
```

```
            principle is transferred in
            approved and
            deposited into the treasury, returning (_amount - profit) Time
         */
        principle.safeTransferFrom( msg.sender, address(this), _amount );
        principle.approve( address( treasury ), _amount );
        treasury.deposit( _amount, address(principle), profit );

        if ( fee != 0 ) { // fee is transferred to dao
            Time.safeTransfer( DAO, fee );
        }
        require(balanceBefore.add(profit) == Time.balanceOf(address(this)), "Not enough Time to cover profit");
        // total debt is increased
        totalDebt = totalDebt.add( value );

        // depositor info is stored
        bondInfo[ _depositor ] = Bond({
            payout: bondInfo[ _depositor ].payout.add( payout ),
            vesting: terms.vestingTerm,
            lastTime: uint32(block.timestamp),
            pricePaid: priceInUSD
        });

        // indexed events are emitted
        emit BondCreated( _amount, payout, block.timestamp.add( terms.vestingTerm ), priceInUSD );
        emit BondPriceChanged( bondPriceInUSD(), _bondPrice(), debtRatio() );

        adjust(); // control variable is adjusted
        return payout;
    }

    /**
     *  @notice redeem bond for user
     *  @param _recipient address
     *  @param _stake bool
     *  @return uint
     */
    function redeem( address _recipient, bool _stake ) external returns ( uint ) {
        require(msg.sender == _recipient, "NA");
        Bond memory info = bondInfo[ _recipient ];
        // (seconds since last interaction / vesting term remaining)
        uint percentVested = percentVestedFor( _recipient );

        if ( percentVested >= 10000 ) { // if fully vested
            delete bondInfo[ _recipient ]; // delete user info
            emit BondRedeemed( _recipient, info.payout, 0 ); // emit bond data
            return stakeOrSend( _recipient, _stake, info.payout ); // pay user everything due

        } else { // if unfinished
            // calculate payout vested
```

```
            principle is transferred in
            approved and
            deposited into the treasury, returning (_amount - profit) Time
         */
        principle.safeTransferFrom( msg.sender, address(this), _amount );
        principle.approve( address( treasury ), _amount );
        treasury.deposit( _amount, address(principle), profit );

        if ( fee != 0 ) { // fee is transferred to dao
            Time.safeTransfer( DAO, fee );
        }
        require(balanceBefore.add(payout) == Time.balanceOf(address(this)), "Not enough Time to cover profit");
        // total debt is increased
        totalDebt = totalDebt.add( value );

        // depositor info is stored
        bondInfo[ _depositor ] = Bond({
            payout: bondInfo[ _depositor ].payout.add( payout ),
            vesting: terms.vestingTerm,
            lastTime: uint32(block.timestamp),
            pricePaid: priceInUSD
        });

        // indexed events are emitted
        emit BondCreated( _amount, payout, block.timestamp.add( terms.vestingTerm ), priceInUSD );
        emit BondPriceChanged( bondPriceInUSD(), _bondPrice(), debtRatio() );

        adjust(); // control variable is adjusted
        return payout;
    }

    /**
     *  @notice redeem bond for user
     *  @param _recipient address
     *  @param _stake bool
     *  @return uint
     */
    function redeem( address _recipient, bool _stake ) external returns ( uint ) {
        require(msg.sender == _recipient, "NA");
        Bond memory info = bondInfo[ _recipient ];
        // (seconds since last interaction / vesting term remaining)
        uint percentVested = percentVestedFor( _recipient );

        if ( percentVested >= 10000 ) { // if fully vested
            delete bondInfo[ _recipient ]; // delete user info
            emit BondRedeemed( _recipient, info.payout, 0 ); // emit bond data
            return stakeOrSend( _recipient, _stake, info.payout ); // pay user everything due

        } else { // if unfinished
            // calculate payout vested
            uint payout = info.payout.mul( percentVested ) / 10000 ;
            // store updated deposit info
            bondInfo[ _recipient ] = Bond({
                payout: info.payout.sub( payout ),
                vesting: info.vesting.sub32( uint32( block.timestamp ).sub32( info.lastTime ) ),
                lastTime: uint32(block.timestamp),
                pricePaid: info.pricePaid
            });

            emit BondRedeemed( _recipient, payout, bondInfo[ _recipient ].payout );
            return stakeOrSend( _recipient, _stake, payout );
        }
    }




    /* ======== INTERNAL HELPER FUNCTIONS ======== */

    /**
     *  @notice allow user to stake payout automatically
     *  @param _stake bool
     *  @param _amount uint
     *  @return uint
     */
    function stakeOrSend( address _recipient, bool _stake, uint _amount ) internal returns ( uint ) {
        if ( !_stake ) { // if user does not want to stake
            Time.transfer( _recipient, _amount ); // send payout
        } else { // if user wants to stake
            if ( useHelper ) { // use if staking warmup is 0
                Time.approve( address(stakingHelper), _amount );
                stakingHelper.stake( _amount, _recipient );
            } else {
                Time.approve( address(staking), _amount );
                staking.stake( _amount, _recipient );
            }
        }
        return _amount;
    }

    /**
     *  @notice makes incremental adjustment to control variable
     */
    function adjust() internal {
        uint timeCanAdjust = adjustment.lastTime.add32( adjustment.buffer );
        if( adjustment.rate != 0 && block.timestamp >= timeCanAdjust ) {
            uint initial = terms.controlVariable;
            uint bcv = initial;
            if ( adjustment.add ) {
                bcv = bcv.add(adjustment.rate);
                if ( bcv >= adjustment.target ) {
                    adjustment.rate = 0;
```

```
            uint payout = info.payout.mul( percentVested ) / 10000 ;
            // store updated deposit info
            bondInfo[ _recipient ] = Bond({
                payout: info.payout.sub( payout ),
                vesting: info.vesting.sub32( uint32( block.timestamp ).sub32( info.lastTime ) ),
                lastTime: uint32(block.timestamp),
                pricePaid: info.pricePaid
            });

            emit BondRedeemed( _recipient, payout, bondInfo[ _recipient ].payout );
            return stakeOrSend( _recipient, _stake, payout );
        }
    }




    /* ======== INTERNAL HELPER FUNCTIONS ======== */

    /**
     *  @notice allow user to stake payout automatically
     *  @param _stake bool
     *  @param _amount uint
     *  @return uint
     */
    function stakeOrSend( address _recipient, bool _stake, uint _amount ) internal returns ( uint ) {
        if ( !_stake ) { // if user does not want to stake
            Time.transfer( _recipient, _amount ); // send payout
        } else { // if user wants to stake
            if ( useHelper ) { // use if staking warmup is 0
                Time.approve( address(stakingHelper), _amount );
                stakingHelper.stake( _amount, _recipient );
            } else {
                Time.approve( address(staking), _amount );
                staking.stake( _amount, _recipient );
            }
        }
        return _amount;
    }

    /**
     *  @notice makes incremental adjustment to control variable
     */
    function adjust() internal {
        uint timeCanAdjust = adjustment.lastTime.add32( adjustment.buffer );
        if( adjustment.rate != 0 && block.timestamp >= timeCanAdjust ) {
            uint initial = terms.controlVariable;
            uint bcv = initial;
            if ( adjustment.add ) {
                bcv = bcv.add(adjustment.rate);
                if ( bcv >= adjustment.target ) {
                    adjustment.rate = 0;
                    bcv = adjustment.target;
                }
            } else {
                bcv = bcv.sub(adjustment.rate);
                if ( bcv <= adjustment.target ) {
                    adjustment.rate = 0;
                    bcv = adjustment.target;
                }
            }
            terms.controlVariable = bcv;
            adjustment.lastTime = uint32(block.timestamp);
            emit ControlVariableAdjustment( initial, bcv, adjustment.rate, adjustment.add );
        }
    }

    /**
     *  @notice reduce total debt
     */
    function decayDebt() internal {
        totalDebt = totalDebt.sub( debtDecay() );
        lastDecay = uint32(block.timestamp);
    }




    /* ======== VIEW FUNCTIONS ======== */

    /**
     *  @notice determine maximum bond size
     *  @return uint
     */
    function maxPayout() public view returns ( uint ) {
        return Time.totalSupply().mul( terms.maxPayout ) / 100000 ;
    }

    /**
     *  @notice calculate interest due for new bond
     *  @param _value uint
     *  @return uint
     */
    function payoutFor( uint _value ) public view returns ( uint ) {
        return FixedPoint.fraction( _value, bondPrice() ).decode112with18() / 1e16 ;
    }


    /**
     *  @notice calculate current bond premium
     *  @return price_ uint
     */
    function bondPrice() public view returns ( uint price_ ) {
        price_ = terms.controlVariable.mul( debtRatio() ).add( 1000000000 ) / 1e7;
        if ( price_ < terms.minimumPrice ) {
            price_ = terms.minimumPrice;
        }
    }

    /**
     *  @notice calculate current bond price and remove floor if above
     *  @return price_ uint
     */
```

```
                    bcv = adjustment.target;
                }
            } else {
                bcv = bcv.sub(adjustment.rate);
                if ( bcv <= adjustment.target ) {
                    adjustment.rate = 0;
                    bcv = adjustment.target;
                }
            }
            terms.controlVariable = bcv;
            adjustment.lastTime = uint32(block.timestamp);
            emit ControlVariableAdjustment( initial, bcv, adjustment.rate, adjustment.add );
        }
    }

    /**
     *  @notice reduce total debt
     */
    function decayDebt() internal {
        totalDebt = totalDebt.sub( debtDecay() );
        lastDecay = uint32(block.timestamp);
    }




    /* ======== VIEW FUNCTIONS ======== */

    /**
     *  @notice determine maximum bond size
     *  @return uint
     */
    function maxPayout() public view returns ( uint ) {
        return Time.totalSupply().mul( terms.maxPayout ) / 100000 ;
    }

    /**
     *  @notice calculate interest due for new bond
     *  @param _value uint
     *  @return uint
     */
    function payoutFor( uint _value ) public view returns ( uint ) {
        return FixedPoint.fraction( _value, bondPrice() ).decode112with18() / 1e16 ;
    }


    /**
     *  @notice calculate current bond premium
     *  @return price_ uint
     */
    function bondPrice() public view returns ( uint price_ ) {
        price_ = terms.controlVariable.mul( debtRatio() ).add( 1000000000 ) / 1e7;
        if ( price_ < terms.minimumPrice ) {
            price_ = terms.minimumPrice;
        }
    }

    /**
     *  @notice calculate current bond price and remove floor if above
     *  @return price_ uint
     */
    function _bondPrice() internal returns ( uint price_ ) {
        price_ = terms.controlVariable.mul( debtRatio() ).add( 1000000000 ) / 1e7;
        if ( price_ < terms.minimumPrice ) {
            price_ = terms.minimumPrice;
        } else if ( terms.minimumPrice != 0 ) {
            terms.minimumPrice = 0;
        }
    }

    /**
     *  @notice converts bond price to DAI value
     *  @return price_ uint
     */
    function bondPriceInUSD() public view returns ( uint price_ ) {
        if( isLiquidityBond ) {
            price_ = bondPrice().mul( bondCalculator.markdown( address(principle) ) ) / 100 ;
        } else {
            price_ = bondPrice().mul( 10 ** principle.decimals() ) / 100;
        }
    }


    /**
     *  @notice calculate current ratio of debt to Time supply
     *  @return debtRatio_ uint
     */
    function debtRatio() public view returns ( uint debtRatio_ ) {
        uint supply = Time.totalSupply();
        debtRatio_ = FixedPoint.fraction(
            currentDebt().mul( 1e9 ),
            supply
        ).decode112with18() / 1e18;
    }

    /**
     *  @notice debt ratio in same terms for reserve or liquidity bonds
     *  @return uint
     */
    function standardizedDebtRatio() external view returns ( uint ) {
        if ( isLiquidityBond ) {
            return debtRatio().mul( bondCalculator.markdown( address(principle) ) ) / 1e9;
        } else {
            return debtRatio();
        }
    }

    /**
     *  @notice calculate debt factoring in decay
     *  @return uint
     */
    function currentDebt() public view returns ( uint ) {
        return totalDebt.sub( debtDecay() );
    }

    /**
     *  @notice amount to decay total debt by
     *  @return decay_ uint
     */
```

```
    function _bondPrice() internal returns ( uint price_ ) {
        price_ = terms.controlVariable.mul( debtRatio() ).add( 1000000000 ) / 1e7;
        if ( price_ < terms.minimumPrice ) {
            price_ = terms.minimumPrice;
        } else if ( terms.minimumPrice != 0 ) {
            terms.minimumPrice = 0;
        }
    }

    /**
     *  @notice converts bond price to DAI value
     *  @return price_ uint
     */
    function bondPriceInUSD() public view returns ( uint price_ ) {
        if( isLiquidityBond ) {
            price_ = bondPrice().mul( bondCalculator.markdown( address(principle) ) ) / 100 ;
        } else {
            price_ = bondPrice().mul( 10 ** principle.decimals() ) / 100;
        }
    }


    /**
     *  @notice calculate current ratio of debt to Time supply
     *  @return debtRatio_ uint
     */
    function debtRatio() public view returns ( uint debtRatio_ ) {
        uint supply = Time.totalSupply();
        debtRatio_ = FixedPoint.fraction(
            currentDebt().mul( 1e9 ),
            supply
        ).decode112with18() / 1e18;
    }

    /**
     *  @notice debt ratio in same terms for reserve or liquidity bonds
     *  @return uint
     */
    function standardizedDebtRatio() external view returns ( uint ) {
        if ( isLiquidityBond ) {
            return debtRatio().mul( bondCalculator.markdown( address(principle) ) ) / 1e9;
        } else {
            return debtRatio();
        }
    }

    /**
     *  @notice calculate debt factoring in decay
     *  @return uint
     */
    function currentDebt() public view returns ( uint ) {
        return totalDebt.sub( debtDecay() );
    }

    /**
     *  @notice amount to decay total debt by
     *  @return decay_ uint
     */
    function debtDecay() public view returns ( uint decay_ ) {
        uint32 timeSinceLast = uint32(block.timestamp).sub32( lastDecay );
        decay_ = totalDebt.mul( timeSinceLast ) / terms.vestingTerm;
        if ( decay_ > totalDebt ) {
            decay_ = totalDebt;
        }
    }


    /**
     *  @notice calculate how far into vesting a depositor is
     *  @param _depositor address
     *  @return percentVested_ uint
     */
    function percentVestedFor( address _depositor ) public view returns ( uint percentVested_ ) {
        Bond memory bond = bondInfo[ _depositor ];
        uint secondsSinceLast = uint32(block.timestamp).sub32( bond.lastTime );
        uint vesting = bond.vesting;

        if ( vesting > 0 ) {
            percentVested_ = secondsSinceLast.mul( 10000 ) / vesting;
        } else {
            percentVested_ = 0;
        }
    }

    /**
     *  @notice calculate amount of Time available for claim by depositor
     *  @param _depositor address
     *  @return pendingPayout_ uint
     */
    function pendingPayoutFor( address _depositor ) external view returns ( uint pendingPayout_ ) {
        uint percentVested = percentVestedFor( _depositor );
        uint payout = bondInfo[ _depositor ].payout;

        if ( percentVested >= 10000 ) {
            pendingPayout_ = payout;
        } else {
            pendingPayout_ = payout.mul( percentVested ) / 10000;
        }
    }




    /* ======= AUXILLIARY ======= */

    /**
     *  @notice allow anyone to send lost tokens (excluding principle or Time) to the DAO
     *  @return bool
     */
    function recoverLostToken(IERC20 _token ) external returns ( bool ) {
        require( _token != Time, "NAT" );
        require( _token != principle, "NAP" );
        uint balance = _token.balanceOf( address(this));
```

```
    function debtDecay() public view returns ( uint decay_ ) {
        uint32 timeSinceLast = uint32(block.timestamp).sub32( lastDecay );
        decay_ = (totalDebt.mul( timeSinceLast )).div(terms.vestingTerm);
        if ( decay_ > totalDebt ) {
            decay_ = totalDebt;
        }
    }


    /**
     *  @notice calculate how far into vesting a depositor is
     *  @param _depositor address
     *  @return percentVested_ uint
     */
    function percentVestedFor( address _depositor ) public view returns ( uint percentVested_ ) {
        Bond memory bond = bondInfo[ _depositor ];
        uint secondsSinceLast = uint32(block.timestamp).sub32( bond.lastTime );
        uint vesting = bond.vesting;

        if ( vesting > 0 ) {
            percentVested_ = secondsSinceLast.mul( 10000 ) / vesting;
        } else {
            percentVested_ = 0;
        }
    }

    /**
     *  @notice calculate amount of Time available for claim by depositor
     *  @param _depositor address
     *  @return pendingPayout_ uint
     */
    function pendingPayoutFor( address _depositor ) external view returns ( uint pendingPayout_ ) {
        uint percentVested = percentVestedFor( _depositor );
        uint payout = bondInfo[ _depositor ].payout;

        if ( percentVested >= 10000 ) {
            pendingPayout_ = payout;
        } else {
            pendingPayout_ = payout.mul( percentVested ) / 10000;
        }
    }




    /* ======= AUXILLIARY ======= */

    /**
     *  @notice allow anyone to send lost tokens (excluding principle or Time) to the DAO
     *  @return bool
     */
    function recoverLostToken(IERC20 _token ) external returns ( bool ) {
        require( _token != Time, "NAT" );
        require( _token != principle, "NAP" );
        uint balance = _token.balanceOf( address(this));
        _token.safeTransfer( DAO,  balance );
        emit LogRecoverLostToken(address(_token), balance);
        return true;
    }
}
```

```
        _token.safeTransfer( DAO,  balance );
        emit LogRecoverLostToken(address(_token), balance);
        return true;
    }

    function recoverLostETH() internal {
        if (address(this).balance > 0) safeTransferETH(DAO, address(this).balance);
    }

    /// @notice Transfers ETH to the recipient address
    /// @dev Fails with `STE`
    /// @param to The destination of the transfer
    /// @param value The value to be transferred
    function safeTransferETH(address to, uint256 value) internal {
        (bool success, ) = to.call{value: value}(new bytes(0));
        require(success, 'STE');
    }
}
```