- 3 Removals + 35 Additions

```
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./PriceOracle.sol";
import "./ComptrollerInterface.sol";
import "./ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./Governance/Comp.sol";

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 */
contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    /// @notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

    /// @notice Emitted when an account enters a market
    event MarketEntered(CToken cToken, address account);

    /// @notice Emitted when an account exits a market
    event MarketExited(CToken cToken, address account);

    /// @notice Emitted when close factor is changed by admin
    event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);

    /// @notice Emitted when a collateral factor is changed by admin
    event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFactorMantissa);

    /// @notice Emitted when liquidation incentive is changed by admin
    event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveMantissa);

    /// @notice Emitted when maxAssets is changed by admin
    event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);

    /// @notice Emitted when price oracle is changed
    event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);

    /// @notice Emitted when pause guardian is changed
```

```
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./PriceOracle.sol";
import "./ComptrollerInterface.sol";
import "./ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./Governance/Comp.sol";

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 */
contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    /// @notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

    /// @notice Emitted when an account enters a market
    event MarketEntered(CToken cToken, address account);

    /// @notice Emitted when an account exits a market
    event MarketExited(CToken cToken, address account);

    /// @notice Emitted when close factor is changed by admin
    event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);

    /// @notice Emitted when a collateral factor is changed by admin
    event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFactorMantissa);

    /// @notice Emitted when liquidation incentive is changed by admin
    event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveMantissa);

    /// @notice Emitted when maxAssets is changed by admin
    event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);

    /// @notice Emitted when price oracle is changed
    event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);

    /// @notice Emitted when pause guardian is changed
```

```
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

    /// @notice Emitted when an action is paused globally
    event ActionPaused(string action, bool pauseState);

    /// @notice Emitted when an action is paused on a market
    event ActionPaused(CToken cToken, string action, bool pauseState);

    /// @notice Emitted when market comped status is changed
    event MarketComped(CToken cToken, bool isComped);

    /// @notice Emitted when COMP rate is changed
    event NewCompRate(uint oldCompRate, uint newCompRate);

    /// @notice Emitted when a new COMP speed is calculated for a market
    event CompSpeedUpdated(CToken indexed cToken, uint newSpeed);

    /// @notice Emitted when COMP is distributed to a supplier
    event DistributedSupplierComp(CToken indexed cToken, address indexed supplier, uint compDelta, uint compSupplyIndex);

    /// @notice Emitted when COMP is distributed to a borrower
    event DistributedBorrowerComp(CToken indexed cToken, address indexed borrower, uint compDelta, uint compBorrowIndex);

    /// @notice The threshold above which the flywheel transfers COMP, in wei
    uint public constant compClaimThreshold = 0.001e18;

    /// @notice The initial COMP index for a market
    uint224 public constant compInitialIndex = 1e36;

    // closeFactorMantissa must be strictly greater than this value
    uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    // No collateralFactorMantissa may exceed this value
    uint internal constant collateralFactorMaxMantissa = 0.9e18; // 0.9

    // liquidationIncentiveMantissa must be no less than this value
    uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0
```

```
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

    /// @notice Emitted when an action is paused globally
    event ActionPaused(string action, bool pauseState);

    /// @notice Emitted when an action is paused on a market
    event ActionPaused(CToken cToken, string action, bool pauseState);

    /// @notice Emitted when market comped status is changed
    event MarketComped(CToken cToken, bool isComped);

    /// @notice Emitted when COMP rate is changed
    event NewCompRate(uint oldCompRate, uint newCompRate);

    /// @notice Emitted when a new COMP speed is calculated for a market
    event CompSpeedUpdated(CToken indexed cToken, uint newSpeed);

    /// @notice Emitted when COMP is distributed to a supplier
    event DistributedSupplierComp(CToken indexed cToken, address indexed supplier, uint compDelta, uint compSupplyIndex);

    /// @notice Emitted when COMP is distributed to a borrower
    event DistributedBorrowerComp(CToken indexed cToken, address indexed borrower, uint compDelta, uint compBorrowIndex);

    /// @notice The threshold above which the flywheel transfers COMP, in wei
    uint public constant compClaimThreshold = 0.001e18;

    /// @notice The initial COMP index for a market
    uint224 public constant compInitialIndex = 1e36;

    // closeFactorMantissa must be strictly greater than this value
    uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    // No collateralFactorMantissa may exceed this value
    uint internal constant collateralFactorMaxMantissa = 1.0e18; // 1.0

    // liquidationIncentiveMantissa must be no less than this value
    uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0
```

```

    // liquidationIncentiveMantissa must be no greater than this value
    uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

    constructor() public {
        admin = msg.sender;
    }

    /*** Assets You Are In ***/

    /**
     * @notice Returns the assets an account has entered
     * @param account The address of the account to pull assets for
     * @return A dynamic list with the assets the account has entered
     */
    function getAssetsIn(address account) external view returns (CToken[] memory) {
        CToken[] memory assetsIn = accountAssets[account];

        return assetsIn;
    }

    /**
     * @notice Returns whether the given account is entered in the given asset
     * @param account The address of the account to check
     * @param cToken The cToken to check
     * @return True if the account is in the asset, otherwise false.
     */
    function checkMembership(address account, CToken cToken) external view returns (bool) {
        return markets[address(cToken)].accountMembership[account];
    }

    /**
     * @notice Add assets to be included in account liquidity calculation
     * @param cTokens The list of addresses of the cToken markets to be enabled
     * @return Success indicator for whether each corresponding market was entered
     */
    function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
        uint len = cTokens.length;

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }

        return results;
    }

    /**
```


```

    // liquidationIncentiveMantissa must be no greater than this value
    uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

    constructor() public {
        admin = msg.sender;
    }

    /*** Assets You Are In ***/

    /**
     * @notice Returns the assets an account has entered
     * @param account The address of the account to pull assets for
     * @return A dynamic list with the assets the account has entered
     */
    function getAssetsIn(address account) external view returns (CToken[] memory) {
        CToken[] memory assetsIn = accountAssets[account];

        return assetsIn;
    }

    /**
     * @notice Returns whether the given account is entered in the given asset
     * @param account The address of the account to check
     * @param cToken The cToken to check
     * @return True if the account is in the asset, otherwise false.
     */
    function checkMembership(address account, CToken cToken) external view returns (bool) {
        return markets[address(cToken)].accountMembership[account];
    }

    /**
     * @notice Add assets to be included in account liquidity calculation
     * @param cTokens The list of addresses of the cToken markets to be enabled
     * @return Success indicator for whether each corresponding market was entered
     */
    function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
        uint len = cTokens.length;

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }

        return results;
    }

    /**
```

```
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }

        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }

        if (accountAssets[borrower].length >= maxAssets)  {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }

        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //  this avoids having to iterate through the list for the most common use cases
        //  that is, only when we need to perform liquidity checks
        //  and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);

        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *  or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
```


```
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }

        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }

        if (accountAssets[borrower].length >= maxAssets)  {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }

        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //  this avoids having to iterate through the list for the most common use cases
        //  that is, only when we need to perform liquidity checks
        //  and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);

        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *  or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
```

```
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }

        Market storage marketToExit = markets[address(cToken)];

        /* Return true if the sender is not already ‘in’ the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender];
        storedList[assetIndex] = storedList[storedList.length - 1];
        storedList.length--;

        emit MarketExited(cToken, msg.sender);

        return uint(Error.NO_ERROR);
    }

    /*** Policy Hooks ***/

    /**
```


```
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }

        Market storage marketToExit = markets[address(cToken)];

        /* Return true if the sender is not already ‘in’ the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender];
        storedList[assetIndex] = storedList[storedList.length - 1];
        storedList.length--;

        emit MarketExited(cToken, msg.sender);

        return uint(Error.NO_ERROR);
    }

    /*** Policy Hooks ***/

    /**
```

```
     * @notice Checks if the account should be allowed to mint tokens in the given market
     * @param cToken The market to verify the mint against
     * @param minter The account which would get the minted tokens
     * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
     * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!mintGuardianPaused[cToken], "mint is paused");

        // Shh - currently unused
        minter;
        mintAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, minter, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates mint and reverts on rejection. May emit logs.
     * @param cToken Asset being minted
     * @param minter The address minting the tokens
     * @param actualMintAmount The amount of the underlying asset being minted
     * @param mintTokens The number of tokens being minted
     */
    function mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens) external {
        // Shh - currently unused
        cToken;
        minter;
        actualMintAmount;
        mintTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
```

```
     * @notice Checks if the account should be allowed to mint tokens in the given market
     * @param cToken The market to verify the mint against
     * @param minter The account which would get the minted tokens
     * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
     * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!mintGuardianPaused[cToken], "mint is paused");

        // Shh - currently unused
        minter;
        mintAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, minter, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates mint and reverts on rejection. May emit logs.
     * @param cToken Asset being minted
     * @param minter The address minting the tokens
     * @param actualMintAmount The amount of the underlying asset being minted
     * @param mintTokens The number of tokens being minted
     */
    function mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens) external {
        // Shh - currently unused
        cToken;
        minter;
        actualMintAmount;
        mintTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
```

```
     * @notice Checks if the account should be allowed to redeem tokens in the given market
     * @param cToken The market to verify the redeem against
     * @param redeemer The account which would redeem the tokens
     * @param redeemTokens The number of cTokens to exchange for the underlying asset in the market
     * @return 0 if the redeem is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uint) {

        uint allowed = redeemAllowedInternal(cToken, redeemer, redeemTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, redeemer, false);

        return uint(Error.NO_ERROR);
    }

    function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens) internal view returns (uint) {
        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
        if (!markets[cToken].accountMembership[redeemer]) {
            return uint(Error.NO_ERROR);
        }

        /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer, CToken(cToken), redeemTokens, 0);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates redeem and reverts on rejection. May emit logs.
     * @param cToken Asset being redeemed
```

```
     * @notice Checks if the account should be allowed to redeem tokens in the given market
     * @param cToken The market to verify the redeem against
     * @param redeemer The account which would redeem the tokens
     * @param redeemTokens The number of cTokens to exchange for the underlying asset in the market
     * @return 0 if the redeem is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uint) {
        // update the asset price
        oracle.updatePrice(CToken(cToken));

        uint allowed = redeemAllowedInternal(cToken, redeemer, redeemTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, redeemer, false);

        return uint(Error.NO_ERROR);
    }

    function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens) internal view returns (uint) {
        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
        if (!markets[cToken].accountMembership[redeemer]) {
            return uint(Error.NO_ERROR);
        }

        /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer, CToken(cToken), redeemTokens, 0);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates redeem and reverts on rejection. May emit logs.
     * @param cToken Asset being redeemed
```

```
     * @param redeemer The address redeeming the tokens
     * @param redeemAmount The amount of the underlying asset being redeemed
     * @param redeemTokens The number of tokens being redeemed
     */
    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
        // Shh - currently unused
        cToken;
        redeemer;

        // Require tokens is zero or amount is also zero
        if (redeemTokens == 0 && redeemAmount > 0) {
            revert("redeemTokens zero");
        }
    }

    /**
     * @notice Checks if the account should be allowed to borrow the underlying asset of the given market
     * @param cToken The market to verify the borrow against
     * @param borrower The account which would borrow the asset
     * @param borrowAmount The amount of underlying the account would borrow
     * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!borrowGuardianPaused[cToken], "borrow is paused");

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (!markets[cToken].accountMembership[borrower]) {
            // only cTokens may call borrowAllowed if borrower not in market
            require(msg.sender == cToken, "sender must be cToken");

            // attempt to add borrower to the market
            Error err = addToMarketInternal(CToken(msg.sender), borrower);
            if (err != Error.NO_ERROR) {
                return uint(err);
            }

            // it should be impossible to break the important invariant
```

```
     * @param redeemer The address redeeming the tokens
     * @param redeemAmount The amount of the underlying asset being redeemed
     * @param redeemTokens The number of tokens being redeemed
     */
    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
        // Shh - currently unused
        cToken;
        redeemer;

        // Require tokens is zero or amount is also zero
        if (redeemTokens == 0 && redeemAmount > 0) {
            revert("redeemTokens zero");
        }
    }

    /**
     * @notice Checks if the account should be allowed to borrow the underlying asset of the given market
     * @param cToken The market to verify the borrow against
     * @param borrower The account which would borrow the asset
     * @param borrowAmount The amount of underlying the account would borrow
     * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!borrowGuardianPaused[cToken], "borrow is paused");

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (!markets[cToken].accountMembership[borrower]) {
            // only cTokens may call borrowAllowed if borrower not in market
            require(msg.sender == cToken, "sender must be cToken");

            // attempt to add borrower to the market
            Error err = addToMarketInternal(CToken(msg.sender), borrower);
            if (err != Error.NO_ERROR) {
                return uint(err);
            }

            // it should be impossible to break the important invariant
```

```
            assert(markets[cToken].accountMembership[borrower]);
        }



        if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
            return uint(Error.PRICE_ERROR);
        }

        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(borrower, CToken(cToken), 0, borrowAmount);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates borrow and reverts on rejection. May emit logs.
     * @param cToken Asset whose underlying is being borrowed
     * @param borrower The address borrowing the underlying
     * @param borrowAmount The amount of the underlying asset requested to borrow
     */
    function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
        // Shh - currently unused
        cToken;
        borrower;
        borrowAmount;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to repay a borrow in the given market
     * @param cToken The market to verify the repay against
     * @param payer The account which would repay the asset
     * @param borrower The account which would borrowed the asset
```

```
            assert(markets[cToken].accountMembership[borrower]);
        }

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
            return uint(Error.PRICE_ERROR);
        }

        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(borrower, CToken(cToken), 0, borrowAmount);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates borrow and reverts on rejection. May emit logs.
     * @param cToken Asset whose underlying is being borrowed
     * @param borrower The address borrowing the underlying
     * @param borrowAmount The amount of the underlying asset requested to borrow
     */
    function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
        // Shh - currently unused
        cToken;
        borrower;
        borrowAmount;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to repay a borrow in the given market
     * @param cToken The market to verify the repay against
     * @param payer The account which would repay the asset
     * @param borrower The account which would borrowed the asset
```

```
     * @param repayAmount The amount of the underlying asset the account would repay
     * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        payer;
        borrower;
        repayAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }


        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates repayBorrow and reverts on rejection. May emit logs.
     * @param cToken Asset being repaid
     * @param payer The address repaying the borrow
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint actualRepayAmount,
        uint borrowerIndex) external {
        // Shh - currently unused
        cToken;
        payer;
        borrower;
        actualRepayAmount;
        borrowerIndex;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the liquidation should be allowed to occur
```

```
     * @param repayAmount The amount of the underlying asset the account would repay
     * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        payer;
        borrower;
        repayAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCompBorrowIndex(cToken, borrowIndex);
        distributeBorrowerComp(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates repayBorrow and reverts on rejection. May emit logs.
     * @param cToken Asset being repaid
     * @param payer The address repaying the borrow
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint actualRepayAmount,
        uint borrowerIndex) external {
        // Shh - currently unused
        cToken;
        payer;
        borrower;
        actualRepayAmount;
        borrowerIndex;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the liquidation should be allowed to occur
```

```
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        liquidator;

        if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* The borrower must have shortfall in order to be liquidatable */
        (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall == 0) {
            return uint(Error.INSUFFICIENT_SHORTFALL);
        }

        /* The liquidator may not repay more than what is allowed by the closeFactor */
        uint borrowBalance = CToken(cTokenBorrowed).borrowBalanceStored(borrower);
        (MathError mathErr, uint maxClose) = mulScalarTruncate(Exp({mantissa: closeFactorMantissa}), borrowBalance);
        if (mathErr != MathError.NO_ERROR) {
            return uint(Error.MATH_ERROR);
        }
        if (repayAmount > maxClose) {
            return uint(Error.TOO_MUCH_REPAY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
```

```
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        liquidator;

        if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // update the asset price
        oracle.updatePrice(CToken(cTokenBorrowed));
        oracle.updatePrice(CToken(cTokenCollateral));

        /* The borrower must have shortfall in order to be liquidatable */
        (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall == 0) {
            return uint(Error.INSUFFICIENT_SHORTFALL);
        }

        /* The liquidator may not repay more than what is allowed by the closeFactor */
        uint borrowBalance = CToken(cTokenBorrowed).borrowBalanceStored(borrower);
        (MathError mathErr, uint maxClose) = mulScalarTruncate(Exp({mantissa: closeFactorMantissa}), borrowBalance);
        if (mathErr != MathError.NO_ERROR) {
            return uint(Error.MATH_ERROR);
        }
        if (repayAmount > maxClose) {
            return uint(Error.TOO_MUCH_REPAY);
        }

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
```

```
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint actualRepayAmount,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenBorrowed;
        cTokenCollateral;
        liquidator;
        borrower;
        actualRepayAmount;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!seizeGuardianPaused, "seize is paused");

        // Shh - currently unused
        seizeTokens;

        if (!markets[cTokenCollateral].isListed || !markets[cTokenBorrowed].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
```

```
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint actualRepayAmount,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenBorrowed;
        cTokenCollateral;
        liquidator;
        borrower;
        actualRepayAmount;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!seizeGuardianPaused, "seize is paused");

        // Shh - currently unused
        seizeTokens;

        if (!markets[cTokenCollateral].isListed || !markets[cTokenBorrowed].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
```

```
            return uint(Error.COMPTROLLER_MISMATCH);
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cTokenCollateral);
        distributeSupplierComp(cTokenCollateral, borrower, false);
        distributeSupplierComp(cTokenCollateral, liquidator, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenCollateral;
        cTokenBorrowed;
        liquidator;
        borrower;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
```

```
            return uint(Error.COMPTROLLER_MISMATCH);
        }

        // update the asset price
        oracle.updatePrice(CToken(cTokenCollateral));
        oracle.updatePrice(CToken(cTokenBorrowed));

        // Keep the flywheel moving
        updateCompSupplyIndex(cTokenCollateral);
        distributeSupplierComp(cTokenCollateral, borrower, false);
        distributeSupplierComp(cTokenCollateral, liquidator, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenCollateral;
        cTokenBorrowed;
        liquidator;
        borrower;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
```

```
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");


        // Currently the only consideration is whether or not
        //  the src is allowed to redeem this many tokens
        uint allowed = redeemAllowedInternal(cToken, src, transferTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, src, false);
        distributeSupplierComp(cToken, dst, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates transfer and reverts on rejection. May emit logs.
     * @param cToken Asset being transferred
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     */
    function transferVerify(address cToken, address src, address dst, uint transferTokens) external {
        // Shh - currently unused
        cToken;
        src;
        dst;
        transferTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /*** Liquidity/Liquidation Calculations ***/

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
```

```
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // update the asset price
        oracle.updatePrice(CToken(cToken));

        // Currently the only consideration is whether or not
        //  the src is allowed to redeem this many tokens
        uint allowed = redeemAllowedInternal(cToken, src, transferTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCompSupplyIndex(cToken);
        distributeSupplierComp(cToken, src, false);
        distributeSupplierComp(cToken, dst, false);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates transfer and reverts on rejection. May emit logs.
     * @param cToken Asset being transferred
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     */
    function transferVerify(address cToken, address src, address dst, uint transferTokens) external {
        // Shh - currently unused
        cToken;
        src;
        dst;
        transferTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /*** Liquidity/Liquidation Calculations ***/

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
```

```
     *  Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     *  whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code (semi-opaque),
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);

        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code,
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidityInternal(address account) internal view returns (Error, uint, uint) {
        return getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
```

```
     *  Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     *  whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code (semi-opaque),
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);

        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code,
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidityInternal(address account) internal view returns (Error, uint, uint) {
        return getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
```

```
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @dev Note that we calculate the exchangeRateStored for each collateral cToken using stored data,
     *  without calculating accumulated interest.
     * @return (possible error code,
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidityInternal(
        address account,
        CToken cTokenModify,
        uint redeemTokens,
        uint borrowAmount) internal view returns (Error, uint, uint) {

        AccountLiquidityLocalVars memory vars; // Holds all our calculation results
        uint oErr;
        MathError mErr;

        // For each asset the account is in
        CToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            CToken asset = assets[i];

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) = asset.getAccountSnapshot(account);
```

```
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @dev Note that we calculate the exchangeRateStored for each collateral cToken using stored data,
     *  without calculating accumulated interest.
     * @return (possible error code,
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidityInternal(
        address account,
        CToken cTokenModify,
        uint redeemTokens,
        uint borrowAmount) internal view returns (Error, uint, uint) {

        AccountLiquidityLocalVars memory vars; // Holds all our calculation results
        uint oErr;
        MathError mErr;

        // For each asset the account is in
        CToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            CToken asset = assets[i];

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) = asset.getAccountSnapshot(account);
```

```
        if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
            return (Error.SNAPSHOT_ERROR, 0, 0);
        }
        vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
        vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

        // Get the normalized price of the asset
        vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
        if (vars.oraclePriceMantissa == 0) {
            return (Error.PRICE_ERROR, 0, 0);
        }
        vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

        // Pre-compute a conversion factor from tokens -> ether (normalized price value)
        (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate, vars.oraclePrice);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // sumCollateral += tokensToDenom * cTokenBalance
        (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom, vars.cTokenBalance, vars.sumCollateral);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // sumBorrowPlusEffects += oraclePrice * borrowBalance
        (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, vars.borrowBalance, vars.sumBorrowPlusEffects);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // Calculate effects of interacting with cTokenModify
        if (asset == cTokenModify) {
            // redeem effect
            // sumBorrowPlusEffects += tokensToDenom * redeemTokens
            (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.sumBorrowPlusEffects);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // borrow effect
            // sumBorrowPlusEffects += oraclePrice * borrowAmount
```

```
        if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between upgrades
            return (Error.SNAPSHOT_ERROR, 0, 0);
        }
        vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
        vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

        // Get the normalized price of the asset
        vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
        if (vars.oraclePriceMantissa == 0) {
            return (Error.PRICE_ERROR, 0, 0);
        }
        vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

        // Pre-compute a conversion factor from tokens -> ether (normalized price value)
        (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate, vars.oraclePrice);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // sumCollateral += tokensToDenom * cTokenBalance
        (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom, vars.cTokenBalance, vars.sumCollateral);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // sumBorrowPlusEffects += oraclePrice * borrowBalance
        (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, vars.borrowBalance, vars.sumBorrowPlusEffects);
        if (mErr != MathError.NO_ERROR) {
            return (Error.MATH_ERROR, 0, 0);
        }

        // Calculate effects of interacting with cTokenModify
        if (asset == cTokenModify) {
            // redeem effect
            // sumBorrowPlusEffects += tokensToDenom * redeemTokens
            (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.sumBorrowPlusEffects);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // borrow effect
            // sumBorrowPlusEffects += oraclePrice * borrowAmount
```

```
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }
            }
        }

        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
        }
    }

    /**
     * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
     * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
     * @param cTokenBorrowed The address of the borrowed cToken
     * @param cTokenCollateral The address of the collateral cToken
     * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral tokens
     * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
     */
    function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint actualRepayAmount) external view returns (uint, uint) {
        /* Read oracle prices for borrowed and collateral markets */
        uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
        uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
        if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
            return (uint(Error.PRICE_ERROR), 0);
        }

        /*
         * Get the exchange rate and calculate the number of collateral tokens to seize:
         *  seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
         *  seizeTokens = seizeAmount / exchangeRate
         *   = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
         */
        uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on error
```


```
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }
            }
        }

        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
        }
    }

    /**
     * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
     * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
     * @param cTokenBorrowed The address of the borrowed cToken
     * @param cTokenCollateral The address of the collateral cToken
     * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral tokens
     * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
     */
    function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint actualRepayAmount) external view returns (uint, uint) {
        /* Read oracle prices for borrowed and collateral markets */
        uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
        uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
        if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
            return (uint(Error.PRICE_ERROR), 0);
        }

        /*
         * Get the exchange rate and calculate the number of collateral tokens to seize:
         *  seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
         *  seizeTokens = seizeAmount / exchangeRate
         *   = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
         */
        uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on error
```

```
        uint seizeTokens;
        Exp memory numerator;
        Exp memory denominator;
        Exp memory ratio;
        MathError mathErr;

        (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, denominator) = mulExp(priceCollateralMantissa, exchangeRateMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, ratio) = divExp(numerator, denominator);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, seizeTokens) = mulScalarTruncate(ratio, actualRepayAmount);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        return (uint(Error.NO_ERROR), seizeTokens);
    }

    /*** Admin Functions ***/

    /**
      * @notice Sets a new price oracle for the comptroller
      * @dev Admin function to set a new price oracle
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);

        return uint(Error.NO_ERROR);
    }

    /**
```

```
        uint seizeTokens;
        Exp memory numerator;
        Exp memory denominator;
        Exp memory ratio;
        MathError mathErr;

        (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, denominator) = mulExp(priceCollateralMantissa, exchangeRateMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, ratio) = divExp(numerator, denominator);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, seizeTokens) = mulScalarTruncate(ratio, actualRepayAmount);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        return (uint(Error.NO_ERROR), seizeTokens);
    }

    /*** Admin Functions ***/

    /**
      * @notice Sets a new price oracle for the comptroller
      * @dev Admin function to set a new price oracle
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);

        return uint(Error.NO_ERROR);
    }

    /**
```

```
     * @notice Sets the closeFactor used when liquidating borrows
     * @dev Admin function to set closeFactor
     * @param newCloseFactorMantissa New close factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
        }

        Exp memory newCloseFactorExp = Exp({mantissa: newCloseFactorMantissa});
        Exp memory lowLimit = Exp({mantissa: closeFactorMinMantissa});
        if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        Exp memory highLimit = Exp({mantissa: closeFactorMaxMantissa});
        if (lessThanExp(highLimit, newCloseFactorExp)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Sets the collateralFactor for a market
     * @dev Admin function to set per-market collateralFactor
     * @param cToken The market to set the factor on
     * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

        // Verify market is listed
```

```
     * @notice Sets the closeFactor used when liquidating borrows
     * @dev Admin function to set closeFactor
     * @param newCloseFactorMantissa New close factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
        }

        Exp memory newCloseFactorExp = Exp({mantissa: newCloseFactorMantissa});
        Exp memory lowLimit = Exp({mantissa: closeFactorMinMantissa});
        if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        Exp memory highLimit = Exp({mantissa: closeFactorMaxMantissa});
        if (lessThanExp(highLimit, newCloseFactorExp)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Sets the collateralFactor for a market
     * @dev Admin function to set per-market collateralFactor
     * @param cToken The market to set the factor on
     * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

        // Verify market is listed
```

```
        Market storage market = markets[address(cToken)];
        if (!market.isListed) {
            return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
        }

        Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});

        // Check collateral factor <= 0.9
        Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
        if (lessThanExp(highLimit, newCollateralFactorExp)) {
            return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
        }

        // If collateral factor != 0, fail if price == 0
        if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
            return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
        }

        // Set market's collateral factor to new collateral factor, remember old value
        uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
        market.collateralFactorMantissa = newCollateralFactorMantissa;

        // Emit event with asset, old collateral factor, and new collateral factor
        emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets maxAssets which controls how many markets can be entered
      * @dev Admin function to set maxAssets
      * @param newMaxAssets New max assets
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setMaxAssets(uint newMaxAssets) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
        }

        uint oldMaxAssets = maxAssets;
        maxAssets = newMaxAssets;
        emit NewMaxAssets(oldMaxAssets, newMaxAssets);

        return uint(Error.NO_ERROR);
    }

```

```
        Market storage market = markets[address(cToken)];
        if (!market.isListed) {
            return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
        }

        Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});

        // Check collateral factor <= 0.9
        Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
        if (lessThanExp(highLimit, newCollateralFactorExp)) {
            return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
        }

        // If collateral factor != 0, fail if price == 0
        if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
            return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
        }

        // Set market's collateral factor to new collateral factor, remember old value
        uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
        market.collateralFactorMantissa = newCollateralFactorMantissa;

        // Emit event with asset, old collateral factor, and new collateral factor
        emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Sets maxAssets which controls how many markets can be entered
      * @dev Admin function to set maxAssets
      * @param newMaxAssets New max assets
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setMaxAssets(uint newMaxAssets) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
        }

        uint oldMaxAssets = maxAssets;
        maxAssets = newMaxAssets;
        emit NewMaxAssets(oldMaxAssets, newMaxAssets);

        return uint(Error.NO_ERROR);
    }

```

```
    /**
      * @notice Sets liquidationIncentive
      * @dev Admin function to set liquidationIncentive
      * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
        }

        // Check de-scaled min <= newLiquidationIncentive <= max
        Exp memory newLiquidationIncentive = Exp({mantissa: newLiquidationIncentiveMantissa});
        Exp memory minLiquidationIncentive = Exp({mantissa: liquidationIncentiveMinMantissa});
        if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        Exp memory maxLiquidationIncentive = Exp({mantissa: liquidationIncentiveMaxMantissa});
        if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        // Save current value for use in log
        uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;

        // Set liquidation incentive to new incentive
        liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;

        // Emit event with old incentive, new incentive
        emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Add the market to the markets mapping and set it as listed
      * @dev Admin function to set isListed and add support for the market
      * @param cToken The address of the market (token) to list
      * @return uint 0=success, otherwise a failure. (See enum Error for details)
```

```
    /**
      * @notice Sets liquidationIncentive
      * @dev Admin function to set liquidationIncentive
      * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
        }

        // Check de-scaled min <= newLiquidationIncentive <= max
        Exp memory newLiquidationIncentive = Exp({mantissa: newLiquidationIncentiveMantissa});
        Exp memory minLiquidationIncentive = Exp({mantissa: liquidationIncentiveMinMantissa});
        if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        Exp memory maxLiquidationIncentive = Exp({mantissa: liquidationIncentiveMaxMantissa});
        if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
            return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        // Save current value for use in log
        uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;

        // Set liquidation incentive to new incentive
        liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;

        // Emit event with old incentive, new incentive
        emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);

        return uint(Error.NO_ERROR);
    }

    /**
      * @notice Add the market to the markets mapping and set it as listed
      * @dev Admin function to set isListed and add support for the market
      * @param cToken The address of the market (token) to list
      * @return uint 0=success, otherwise a failure. (See enum Error for details)
```

```
     */
    function _supportMarket(CToken cToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(cToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        cToken.isCToken(); // Sanity check to make sure its really a CToken

        markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});

        _addMarketInternal(address(cToken));

        emit MarketListed(cToken);

        return uint(Error.NO_ERROR);
    }

    function _addMarketInternal(address cToken) internal {
        for (uint i = 0; i < allMarkets.length; i++) {
            require(allMarkets[i] != CToken(cToken), "market already added");
        }
        allMarkets.push(CToken(cToken));
    }

    /**
     * @notice Admin function to change the Pause Guardian
     * @param newPauseGuardian The address of the new Pause Guardian
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
        }

        // Save current value for inclusion in log
        address oldPauseGuardian = pauseGuardian;

        // Store pauseGuardian with value newPauseGuardian
        pauseGuardian = newPauseGuardian;

        // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
        emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

        return uint(Error.NO_ERROR);
    }

```

```
     */
    function _supportMarket(CToken cToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(cToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        cToken.isCToken(); // Sanity check to make sure its really a CToken

        markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});

        _addMarketInternal(address(cToken));

        emit MarketListed(cToken);

        return uint(Error.NO_ERROR);
    }

    function _addMarketInternal(address cToken) internal {
        for (uint i = 0; i < allMarkets.length; i++) {
            require(allMarkets[i] != CToken(cToken), "market already added");
        }
        allMarkets.push(CToken(cToken));
    }

    /**
     * @notice Admin function to change the Pause Guardian
     * @param newPauseGuardian The address of the new Pause Guardian
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
        }

        // Save current value for inclusion in log
        address oldPauseGuardian = pauseGuardian;

        // Store pauseGuardian with value newPauseGuardian
        pauseGuardian = newPauseGuardian;

        // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
        emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

        return uint(Error.NO_ERROR);
    }

```

```
    function _setMintPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        mintGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Mint", state);
        return state;
    }

    function _setBorrowPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        borrowGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Borrow", state);
        return state;
    }

    function _setTransferPaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        transferGuardianPaused = state;
        emit ActionPaused("Transfer", state);
        return state;
    }

    function _setSeizePaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        seizeGuardianPaused = state;
        emit ActionPaused("Seize", state);
        return state;
    }

    function _become(Unitroller unitroller) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        require(unitroller._acceptImplementation() == 0, "change not authorized");
    }

```

```
    function _setMintPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        mintGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Mint", state);
        return state;
    }

    function _setBorrowPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        borrowGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Borrow", state);
        return state;
    }

    function _setTransferPaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        transferGuardianPaused = state;
        emit ActionPaused("Transfer", state);
        return state;
    }

    function _setSeizePaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        seizeGuardianPaused = state;
        emit ActionPaused("Seize", state);
        return state;
    }

    function _become(Unitroller unitroller) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        require(unitroller._acceptImplementation() == 0, "change not authorized");
    }

```

```
    /**
     * @notice Checks caller is admin, or this contract is becoming the new implementation
     */
    function adminOrInitializing() internal view returns (bool) {
        return msg.sender == admin || msg.sender == comptrollerImplementation;
    }

    /*** Comp Distribution ***/

    /**
     * @notice Recalculate and update COMP speeds for all COMP markets
     */
    function refreshCompSpeeds() public {
        require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");
        refreshCompSpeedsInternal();
    }

    function refreshCompSpeedsInternal() internal {
        CToken[] memory allMarkets_ = allMarkets;

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
            updateCompSupplyIndex(address(cToken));
            updateCompBorrowIndex(address(cToken), borrowIndex);
        }

        Exp memory totalUtility = Exp({mantissa: 0});
        Exp[] memory utilities = new Exp[](allMarkets_.length);
        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            if (markets[address(cToken)].isComped) {

                Exp memory assetPrice = Exp({mantissa: oracle.getUnderlyingPrice(cToken)});
                Exp memory utility = mul_(assetPrice, cToken.totalBorrows());
                utilities[i] = utility;
                totalUtility = add_(totalUtility, utility);
            }
        }

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets[i];
            uint newSpeed = totalUtility.mantissa > 0 ? mul_(compRate, div_(utilities[i], totalUtility)) : 0;
            compSpeeds[address(cToken)] = newSpeed;
```

```
    /**
     * @notice Checks caller is admin, or this contract is becoming the new implementation
     */
    function adminOrInitializing() internal view returns (bool) {
        return msg.sender == admin || msg.sender == comptrollerImplementation;
    }

    /*** Comp Distribution ***/

    /**
     * @notice Recalculate and update COMP speeds for all COMP markets
     */
    function refreshCompSpeeds() public {
        require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");
        refreshCompSpeedsInternal();
    }

    function refreshCompSpeedsInternal() internal {
        CToken[] memory allMarkets_ = allMarkets;

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
            updateCompSupplyIndex(address(cToken));
            updateCompBorrowIndex(address(cToken), borrowIndex);
        }

        Exp memory totalUtility = Exp({mantissa: 0});
        Exp[] memory utilities = new Exp[](allMarkets_.length);
        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets_[i];
            if (markets[address(cToken)].isComped) {
                oracle.updatePrice(cToken);
                Exp memory assetPrice = Exp({mantissa: oracle.getUnderlyingPrice(cToken)});
                Exp memory utility = mul_(assetPrice, cToken.totalBorrows());
                utilities[i] = utility;
                totalUtility = add_(totalUtility, utility);
            }
        }

        for (uint i = 0; i < allMarkets_.length; i++) {
            CToken cToken = allMarkets[i];
            uint newSpeed = totalUtility.mantissa > 0 ? mul_(compRate, div_(utilities[i], totalUtility)) : 0;
            compSpeeds[address(cToken)] = newSpeed;
```

```
            emit CompSpeedUpdated(cToken, newSpeed);
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the supply index
     * @param cToken The market whose supply index to update
     */
    function updateCompSupplyIndex(address cToken) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        uint supplySpeed = compSpeeds[cToken];
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
        if (deltaBlocks > 0 && supplySpeed > 0) {
            uint supplyTokens = CToken(cToken).totalSupply();
            uint compAccrued = mul_(deltaBlocks, supplySpeed);
            Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens) : Double({mantissa: 0});
            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
            compSupplyState[cToken] = CompMarketState({
                index: safe224(index.mantissa, "new index exceeds 224 bits"),
                block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the borrow index
     * @param cToken The market whose borrow index to update
     */
    function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
        CompMarketState storage borrowState = compBorrowState[cToken];
        uint borrowSpeed = compSpeeds[cToken];
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
        if (deltaBlocks > 0 && borrowSpeed > 0) {
            uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
            uint compAccrued = mul_(deltaBlocks, borrowSpeed);
            Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) : Double({mantissa: 0});
```

```
            emit CompSpeedUpdated(cToken, newSpeed);
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the supply index
     * @param cToken The market whose supply index to update
     */
    function updateCompSupplyIndex(address cToken) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        uint supplySpeed = compSpeeds[cToken];
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
        if (deltaBlocks > 0 && supplySpeed > 0) {
            uint supplyTokens = CToken(cToken).totalSupply();
            uint compAccrued = mul_(deltaBlocks, supplySpeed);
            Double memory ratio = supplyTokens > 0 ? fraction(compAccrued, supplyTokens) : Double({mantissa: 0});
            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
            compSupplyState[cToken] = CompMarketState({
                index: safe224(index.mantissa, "new index exceeds 224 bits"),
                block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }

    /**
     * @notice Accrue COMP to the market by updating the borrow index
     * @param cToken The market whose borrow index to update
     */
    function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
        CompMarketState storage borrowState = compBorrowState[cToken];
        uint borrowSpeed = compSpeeds[cToken];
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
        if (deltaBlocks > 0 && borrowSpeed > 0) {
            uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
            uint compAccrued = mul_(deltaBlocks, borrowSpeed);
            Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) : Double({mantissa: 0});
```

```
            Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
            compBorrowState[cToken] = CompMarketState({
                index: safe224(index.mantissa, "new index exceeds 224 bits"),
                block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }

    /**
     * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
     * @param cToken The market in which the supplier is interacting
     * @param supplier The address of the supplier to distribute COMP to
     */
    function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        Double memory supplyIndex = Double({mantissa: supplyState.index});
        Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][supplier]});
        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
            supplierIndex.mantissa = compInitialIndex;
        }

        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
        uint supplierTokens = CToken(cToken).balanceOf(supplier);
        uint supplierDelta = mul_(supplierTokens, deltaIndex);
        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
        compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ? 0 : compClaimThreshold);
        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
    }

    /**
     * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
     * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
     * @param cToken The market in which the borrower is interacting
     * @param borrower The address of the borrower to distribute COMP to
```


```
            Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
            compBorrowState[cToken] = CompMarketState({
                index: safe224(index.mantissa, "new index exceeds 224 bits"),
                block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }

    /**
     * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
     * @param cToken The market in which the supplier is interacting
     * @param supplier The address of the supplier to distribute COMP to
     */
    function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
        CompMarketState storage supplyState = compSupplyState[cToken];
        Double memory supplyIndex = Double({mantissa: supplyState.index});
        Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][supplier]});
        compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
            supplierIndex.mantissa = compInitialIndex;
        }

        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
        uint supplierTokens = CToken(cToken).balanceOf(supplier);
        uint supplierDelta = mul_(supplierTokens, deltaIndex);
        uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
        compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ? 0 : compClaimThreshold);
        emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
    }

    /**
     * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
     * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
     * @param cToken The market in which the borrower is interacting
     * @param borrower The address of the borrower to distribute COMP to
```

```
     */
    function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex, bool distributeAll) internal {
        CompMarketState storage borrowState = compBorrowState[cToken];
        Double memory borrowIndex = Double({mantissa: borrowState.index});
        Double memory borrowerIndex = Double({mantissa: compBorrowerIndex[cToken][borrower]});
        compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

        if (borrowerIndex.mantissa > 0) {
            Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
            uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowIndex);
            uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
            uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
            compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : compClaimThreshold);
            emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
        }
    }

    /**
     * @notice Transfer COMP to the user, if they are above the threshold
     * @dev Note: If there is not enough COMP, we do not perform the transfer all.
     * @param user The address of the user to transfer COMP to
     * @param userAccrued The amount of COMP to (possibly) transfer
     * @return The amount of COMP which was NOT transferred to the user
     */
    function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
        if (userAccrued >= threshold && userAccrued > 0) {
            Comp comp = Comp(getCompAddress());
            uint compRemaining = comp.balanceOf(address(this));
            if (userAccrued <= compRemaining) {
                comp.transfer(user, userAccrued);

                return 0;
            }
        }
        return userAccrued;
    }

    /**
     * @notice Claim all the comp accrued by holder in all markets
     * @param holder The address to claim COMP for
     */
    function claimComp(address holder) public {
```

```
     */
    function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex, bool distributeAll) internal {
        CompMarketState storage borrowState = compBorrowState[cToken];
        Double memory borrowIndex = Double({mantissa: borrowState.index});
        Double memory borrowerIndex = Double({mantissa: compBorrowerIndex[cToken][borrower]});
        compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

        if (borrowerIndex.mantissa > 0) {
            Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
            uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowIndex);
            uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
            uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
            compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : compClaimThreshold);
            emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
        }
    }

    /**
     * @notice Transfer COMP to the user, if they are above the threshold
     * @dev Note: If there is not enough COMP, we do not perform the transfer all.
     * @param user The address of the user to transfer COMP to
     * @param userAccrued The amount of COMP to (possibly) transfer
     * @return The amount of COMP which was NOT transferred to the user
     */
    function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
        // if (userAccrued >= threshold && userAccrued > 0) {
        //     Comp comp = Comp(getCompAddress());
        //     uint compRemaining = comp.balanceOf(address(this));
        //     if (userAccrued <= compRemaining) {
        //         comp.transfer(user, userAccrued);
        //         return 0;
        //     }
        // }
        return userAccrued;
    }

    /**
     * @notice Claim all the comp accrued by holder in all markets
     * @param holder The address to claim COMP for
     */
    function claimComp(address holder) public {
```

```
        return claimComp(holder, allMarkets);
    }

    /**
     * @notice Claim all the comp accrued by holder in the specified markets
     * @param holder The address to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     */
    function claimComp(address holder, CToken[] memory cTokens) public {
        address[] memory holders = new address[](1);
        holders[0] = holder;
        claimComp(holders, cTokens, true, true);
    }

    /**
     * @notice Claim all comp accrued by the holders
     * @param holders The addresses to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     * @param borrowers Whether or not to claim COMP earned by borrowing
     * @param suppliers Whether or not to claim COMP earned by supplying
     */
    function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public {
        for (uint i = 0; i < cTokens.length; i++) {
            CToken cToken = cTokens[i];
            require(markets[address(cToken)].isListed, "market must be listed");
            if (borrowers == true) {
                Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
                updateCompBorrowIndex(address(cToken), borrowIndex);
                for (uint j = 0; j < holders.length; j++) {
                    distributeBorrowerComp(address(cToken), holders[j], borrowIndex, true);
                }
            }
            if (suppliers == true) {
                updateCompSupplyIndex(address(cToken));
                for (uint j = 0; j < holders.length; j++) {
                    distributeSupplierComp(address(cToken), holders[j], true);
                }
            }
        }
    }

    /*** Comp Distribution Admin ***/

    /**
     * @notice Set the amount of COMP distributed per block
```

```
        return claimComp(holder, allMarkets);
    }

    /**
     * @notice Claim all the comp accrued by holder in the specified markets
     * @param holder The address to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     */
    function claimComp(address holder, CToken[] memory cTokens) public {
        address[] memory holders = new address[](1);
        holders[0] = holder;
        claimComp(holders, cTokens, true, true);
    }

    /**
     * @notice Claim all comp accrued by the holders
     * @param holders The addresses to claim COMP for
     * @param cTokens The list of markets to claim COMP in
     * @param borrowers Whether or not to claim COMP earned by borrowing
     * @param suppliers Whether or not to claim COMP earned by supplying
     */
    function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public {
        for (uint i = 0; i < cTokens.length; i++) {
            CToken cToken = cTokens[i];
            require(markets[address(cToken)].isListed, "market must be listed");
            if (borrowers == true) {
                Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
                updateCompBorrowIndex(address(cToken), borrowIndex);
                for (uint j = 0; j < holders.length; j++) {
                    distributeBorrowerComp(address(cToken), holders[j], borrowIndex, true);
                }
            }
            if (suppliers == true) {
                updateCompSupplyIndex(address(cToken));
                for (uint j = 0; j < holders.length; j++) {
                    distributeSupplierComp(address(cToken), holders[j], true);
                }
            }
        }
    }

    /*** Comp Distribution Admin ***/

    /**
     * @notice Set the amount of COMP distributed per block
```

```
     * @param compRate_ The amount of COMP wei per block to distribute
     */
    function _setCompRate(uint compRate_) public {
        require(adminOrInitializing(), "only admin can change comp rate");

        uint oldRate = compRate;
        compRate = compRate_;
        emit NewCompRate(oldRate, compRate_);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Add markets to compMarkets, allowing them to earn COMP in the flywheel
     * @param cTokens The addresses of the markets to add
     */
    function _addCompMarkets(address[] memory cTokens) public {
        require(adminOrInitializing(), "only admin can add comp market");

        for (uint i = 0; i < cTokens.length; i++) {
            _addCompMarketInternal(cTokens[i]);
        }

        refreshCompSpeedsInternal();
    }

    function _addCompMarketInternal(address cToken) internal {
        Market storage market = markets[cToken];
        require(market.isListed == true, "comp market is not listed");
        require(market.isComped == false, "comp market already added");

        market.isComped = true;
        emit MarketComped(CToken(cToken), true);

        if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
            compSupplyState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }

        if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
            compBorrowState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }
    }

    /**
```

```
     * @param compRate_ The amount of COMP wei per block to distribute
     */
    function _setCompRate(uint compRate_) public {
        require(adminOrInitializing(), "only admin can change comp rate");

        uint oldRate = compRate;
        compRate = compRate_;
        emit NewCompRate(oldRate, compRate_);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Add markets to compMarkets, allowing them to earn COMP in the flywheel
     * @param cTokens The addresses of the markets to add
     */
    function _addCompMarkets(address[] memory cTokens) public {
        require(adminOrInitializing(), "only admin can add comp market");

        for (uint i = 0; i < cTokens.length; i++) {
            _addCompMarketInternal(cTokens[i]);
        }

        refreshCompSpeedsInternal();
    }

    function _addCompMarketInternal(address cToken) internal {
        Market storage market = markets[cToken];
        require(market.isListed == true, "comp market is not listed");
        require(market.isComped == false, "comp market already added");

        market.isComped = true;
        emit MarketComped(CToken(cToken), true);

        if (compSupplyState[cToken].index == 0 && compSupplyState[cToken].block == 0) {
            compSupplyState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }

        if (compBorrowState[cToken].index == 0 && compBorrowState[cToken].block == 0) {
            compBorrowState[cToken] = CompMarketState({
                index: compInitialIndex,
                block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }
    }

    /**
```

```
     * @notice Remove a market from compMarkets, preventing it from earning COMP in the flywheel
     * @param cToken The address of the market to drop
     */
    function _dropCompMarket(address cToken) public {
        require(msg.sender == admin, "only admin can drop comp market");

        Market storage market = markets[cToken];
        require(market.isComped == true, "market is not a comp market");

        market.isComped = false;
        emit MarketComped(CToken(cToken), false);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Return all of the markets
     * @dev The automatic getter may be used to access an individual market.
     * @return The list of market addresses
     */
    function getAllMarkets() public view returns (CToken[] memory) {
        return allMarkets;
    }

    function getBlockNumber() public view returns (uint) {
        return block.number;
    }

    /**
     * @notice Return the address of the COMP token
     * @return The address of COMP
     */
    function getCompAddress() public view returns (address) {
        return 0xc00e94Cb662C3520282E6f5717214004A7f26888;
    }
}

```

```
     * @notice Remove a market from compMarkets, preventing it from earning COMP in the flywheel
     * @param cToken The address of the market to drop
     */
    function _dropCompMarket(address cToken) public {
        require(msg.sender == admin, "only admin can drop comp market");

        Market storage market = markets[cToken];
        require(market.isComped == true, "market is not a comp market");

        market.isComped = false;
        emit MarketComped(CToken(cToken), false);

        refreshCompSpeedsInternal();
    }

    /**
     * @notice Return all of the markets
     * @dev The automatic getter may be used to access an individual market.
     * @return The list of market addresses
     */
    function getAllMarkets() public view returns (CToken[] memory) {
        return allMarkets;
    }

    function getBlockNumber() public view returns (uint) {
        return block.number;
    }

    /**
     * @notice Return the address of the COMP token
     * @return The address of COMP
     */
    function getCompAddress() public view returns (address) {
        return 0x0Ed0Ca6872073E02cd3aE005BaF04bA43BE947fA;
    }
}

```