Compound Protocol

[**Final**] Version 2.1

Constants

Constant	Description
liquidationIncentive	A multiplier representing the excess percent value that a user calling liquidate receives, e.g. 1.05 for a 5% bonus. This discount applies to the asset being seized (that is, what was used as collateral).
collateralFactor	A multiplier represtsenting the amount you can borrow against your collateral, e.g. .9 to allow borrowing 90% of collateral value. Must be between 0 and 1.
closeFactor	A number greater than 0.05 and less than or equal to 0.9 which is multiplied by a given asset's *borrowCurrent* to calculate the maximum *repayAmount* when liquidating a borrow for an underwater account.
maxAssets	The max number of assets a single account can participate in (borrow or use as collateral). This does not affect accounts which mint, redeem or transfer without borrowing.
reserveFactor	The portion of accrued interest that goes into reserves, between [0, 1], and likely below 0.10.

Key Terms

Value	Description
borrowCurrent	The user's borrow of a given asset, including accrued interest as of the current block. This is the user's stored principal, times the market's current interest index divided by the user's stored interest index.
sumCollateral	The collateral value of a user's supplied assets, including accrued interest, in terms of ether. This is the sum over all assets of user's token balance, times the (stored) underlying exchange rate of that token, times the value of that asset in terms of eth, times the the asset's *collateralFactor*. *Note: we use the stored exchange rate here, instead of calculating a new exchange rate for each collateral asset. The delta should be small*

	and this is only used for account liquidity checks.
sumBorrow	The value of a users borrowed assets, including accrued interest, in Ether. That is, the sum of borrowCurrent for each of a user's assets.
accountLiquidity	The sumCollateral value of a user's account, denominated in Ether, minus the sumBorrow (for the case where sumCollateral ≥ sumBorrow). This number may be below zero for unhealthy accounts.
maxCloseValue	A user's borrow balance in a given asset, multiplied by the closeFactor; how much value can be repaid by a liquidator.
seizeTokens	The number of cTokens to transfer from the liquidated user to the liquidator. This is the *seizeAmount*, times the liquidation incentive, times the ratio of the oracle prices for the given asset pair, divided by the exchange rate of the collateral asset.
totalBorrowBalance	The total borrow balance, with accrued interest, across all accounts in a money market, as of the current block. *Note: totalBorrowBalance will be strictly larger than totalBorrows from v1, so the result of interest rate calculations will be different if we use the same interest rate models.*
$\text{assets}_{\text{account}}$	A set of markets that an account has participated in with a max size of maxAssets
blocks	When calculating simple interest, blocks refers to the number of blocks that have elapsed since the last time the interest index was calculated. That most recent block is stored as $interestBlockNumber_{asset}$ and is stored whenever an interest index is stored, and blocks is calculated as the current block's number minus $interestBlockNumber_{asset}$.
rate	When calculating simple interest, rate refers to the current interest rate of the market. This was the previous rate that's been "in effect" for blocks blocks.
Exchange Rate Stored	The last stored exchange rate for cTokens to underlying assets. This value does not include borrowing interest since the last interest accrual in this market.
Exchange Rate Current	The current exchange rate (including all trued up borrowing interest) between cTokens and the underlying asset.

Exceptional States

We assume that in *any* error condition, either a) the protocol exits gracefully with an event describing the error if no side-effects have yet occurred, or b) the transaction fails completely. Any exceptions to this rule are noted, except as below.
A number of functions are split into two commands: accrue interest and a *fresh* action. The goal is to separate two discrete events which should occur. First, every time we accrue interest for a market, we help true up balances (and turn simple interest into compound interest) for that market. Second, the fresh functions are only correct if the market's interest has been fully accumulated as of this block. With updated interest, however, these fresh functions have divergent concerns (often, significantly unrelated to interest accrual). We thus build these functions as the sum of two actions to simplify the understanding and modelling of these two separate acts. In practice, this means that even when a method fails gracefully, the transaction may still accrue interest for that market (this is a good thing!).

Interest Rate Model Contract

For each asset, there is an interest index. We effectively track the growth of principal of an arbitrary account over time. We use the ratio of that account's interest versus initial principal to calculate the growth of any given account's interest over a subset of that time interval. The interest model contract specifies the simple interest rate at any moment (which, when compounded for each transaction becomes compound interest). We force this interest rate model to be a pure function over the cash, borrows, and reserves of an asset in the market. For more information, see **Interest Index Calculation Appendix**.

borrowRate(address **cToken,** amount **cash,** amount **borrows,** amount **reserves) returns** uint

- Return the current interest rate for the market
- Note: cToken is the Compound cToken contract, not the underlying asset address.

Price Oracle Contract

The Compound Protocol uses prices from a smart contract called a price oracle. The Comptroller and Liquidate Borrow functions reference the prices in this oracle. Multiple oracles may exist for the different Compound markets.

getUnderlyingPrice(CToken **cToken) returns** uint

- Return price of the underlying asset (as a mantissa)

cToken Contract

cTokens act as ERC-20 interfaces and will be the primary location where users interact with the Compound Protocol. When a user mints, redeems, borrows, repays borrows, liquidates a borrow, or transfers cTokens, she will do so on the cToken contract itself. The only actions that a user performs on another contract are entering and exiting assets via the Comptroller (see below for the **Comptroller contract**).

cTokens each reference an *underlying*. This is usually the underlying ERC-20 contract, though it may be Ether itself or a complex asset. cTokens are the ultimate holders of that underlying asset balance and each call to take in or send out assets originates in the cToken contract. Initially, cEth (Compound Ether token) will be a unique asset, interacting with Ether instead of ERC-20 assets.

Note about cToken Money Markets

The Money Market was the core monolith of the Compound Protocol in the first version. The functions that used to exist in the Money Market now exist in the cTokens, and the old *Market* struct is flattened in to each cToken Market. Functions related to Policies and Liquidity are deferred to the Comptroller contract.

Market Functions

borrowRatePerBlock()

- Returns the current per-block borrow interest rate mantissa for this cToken

supplyRatePerBlock()

- We calculate the supply rate:
  - $underlying = totalSupply \times exchangeRate$
  - $borrowsPer = totalBorrows \div underlying$
  - $supplyRate = borrowRate \times (1 - reserveFactor) \times borrowsPer$
- Returns the current per-block supply interest rate mantissa for this cToken

Accrue Interest()

We accrue interest and update the borrow index on every operation. This increases compounding, approaching the true value, regardless of whether the rest of the operation succeeds or not.

- $totalCash = invoke\ getCashPrior()$
    - *Note: likely makes an external call*
- We get the interest rate (that was in effect since the last update):
    - $borrowRate = call\ interestModel.borrowRate(this, totalCash, totalBorrows, totalReserves)$
    - $simpleInterestFactor = \Delta blocks \ \times borrowRate$
- We update $borrowIndex$:
    - $borrowIndexNew = \ borrowIndex \times (1 + simpleInterestFactor)$
- We calculate the interest accrued:
    - $interestAccumulated = totalBorrows \times simpleInterestFactor$
- We update $borrows$ and $reserves$:
    - $totalBorrowsNew = totalBorrows + interestAccumulated$
    - $totalReservesNew = totalReserves + interestAccumulated \times reserveFactor$
- We store the updates back to the blockchain:
    - Set $accrualBlockNumber = getBlockNumber()$
    - Set $borrowIndex = borrowIndexNew$
    - Set $totalBorrows = totalBorrowsNew$
    - Set $totalReserves = totalReservesNew$
- Emit an **AccrueInterest** event

[CErc20] Mint(uint mintAmount)

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Mint\ Fresh(msg.sender,\ mintAmount)$

[CEther] Mint() payable

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Mint\ Fresh(msg.sender,\ msg.value)$

[Internal] Mint Fresh(address minter, uint mintAmount)

User supplies assets from her own address into the market and receives a balance of cTokens in exchange.

- Fail if $call\ comptroller.mintAllowed(this,\ minter,\ mintAmount) \ \neq 0$
- Verify market's *block number* equals current block number
- Fail if *invoke checkTransferIn(minter, mintAmount)* fails
- We get the current exchange rate and calculate the number of cTokens to be minted:
    - $exchangeRate = invoke\ Exchange\ Rate\ Stored()$
    - $mintTokens = mintAmount \ \div exchangeRate$

        - Note: divisions are rounded, as necessary, toward zero, thus it is possible to mint 0 tokens
- We calculate the new total supply of cTokens and *minter* token balance:
    - $totalSupplyNew = totalSupply + mintTokens$
        - Fails on overflow
    - $accountTokensNew_{minter} = accountTokens_{minter} + mintTokens$
        - Fails on overflow
- **We have finished calculations**. (If any calculations failed with an error, we have already returned with a failure code). Now we can begin effects.
- We invoke *doTransferIn* for the *minter* and the *mintAmount*
    - Note: The cToken must handle variations between ERC-20 and ETH underlying.
    - On success, the cToken holds an additional *mintAmount* of cash
    - If *doTransferIn* fails despite the fact we checked pre-conditions, we revert because we can't be sure if side effects occurred
- We write previously calculated values into storage:
    - Set $totalSupply = cTokenSupplyNew$
    - Set $accountTokens_{minter} = accountTokensNew_{minter}$
- Emit a **Mint** event with $minter$, $mintAmount$, $mintTokens$
- Emit a **Transfer** event from *this* to *minter*
- $call\ comptroller.mintVerify(this,\ minter,\ mintAmount,\ mintTokens)$

Redeem(uint redeemTokens)

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Redeem\ Fresh(msg.sender,\ redeemTokens,\ 0)$

Redeem Underlying(uint redeemAmount)

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Redeem\ Fresh(msg.sender, 0,\ redeemAmount)$

[Internal] Redeem Fresh(address redeemer, uint redeemTokensIn, uint redeemAmountIn)

User relinquishes cTokens and receives the underlying ERC-20 asset from the protocol into her own wallet.

- $exchangeRate = invoke\ Exchange\ Rate\ Stored()$
- If $redeemTokensIn > 0$:
    - We get the current exchange rate and calculate the amount to be redeemed:
        - $redeemTokens = redeemTokensIn$
        - $redeemAmount = redeemTokensIn \times exchangeRate$
- Else:

    - We get the current exchange rate and calculate the amount to be redeemed:
        - $redeemTokens = redeemAmountIn \div exchangeRate$
        - $redeemAmount = redeemAmountIn$
- Fail if $call\ comptroller.redeemAllowed(this,\ redeemer,\ redeemTokens) \neq 0$
- Verify market's *block number* equals current block number
- We calculate the new total supply and *redeemer* token balance:
    - $totalSupplyNew = totalSupply - redeemTokens$
        - Fails if $redeemTokens > totalSupply$
    - $accountTokensNew_{redeemer} = accountTokens_{redeemer} - redeemTokens$
        - Fails if $redeemTokens > accountTokens_{redeemer}$
- Fail gracefully if protocol has insufficient cash
- **We have finished calculations**. (If any calculations failed with an error, we have already returned with a failure code). Now we can begin side effects.
- We invoke *doTransferOut* for the *redeemer* and the *redeemAmount*
    - Note: The cToken must handle variations between ERC-20 and ETH underlying.
    - On success, the cToken has *redeemAmount* less of cash
    - If *doTransferOut* fails despite the fact we checked pre-conditions, we revert because we can't be sure if side effects occurred
- We write previously calculated values into storage
    - Set $totalSupply = totalSupplyNew$
    - Set $accountTokens_{redeemer} = accountTokensNew_{redeemer}$
- Emit a **Redeem** event with $redeemer$, $redeemAmount$, $redeemTokens$
- Emit a **Transfer** event from *redeemer* to *this*
- $call\ comptroller.redeemVerify(this,\ redeemer,\ redeemAmount,\ redeemTokens)$

Borrow(uint borrowAmount)

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Borrow\ Asset\ Fresh(msg.sender,\ borrowAmount)$

[Internal] BorrowFresh(address borrower, uint borrowAmount)

User borrows assets from the protocol.

- Fail if $call\ comptroller.borrowAllowed(this,\ borrower,\ borrowAmount) \neq 0$
- Verify market's *block number* equals current block number
- We calculate the new borrower and total borrow balances:
    - $accountBorrows = invoke\ Borrow\ Balance\ Stored(borrower)$
    - $accountBorrowsNew = accountBorrows + borrowAmount$
        - Fails on overflow

        - $totalBorrowsNew = totalBorrows + borrowAmount$
            - Fails on overflow
- Fail gracefully if protocol has insufficient cash
- **We have finished calculations**. (If any calculations failed with an error, we have already returned with a failure code). Now we can begin side effects.
- We invoke *doTransferOut* for the *borrower* and the *borrowAmount*
    - Note: The cToken must handle variations between ERC-20 and ETH underlying
    - On success, the cToken has *borrowAmount* less of underlying cash
    - If *doTransferOut* fails despite the fact we checked pre-conditions, we revert because we can't be sure if side effects occurred
- We write the previously calculated values into storage:
    - Set $accountBorrows_{borrower} = \{accountBorrowsNew,\ borrowIndex\}$
    - Set $totalBorrows\ =\ totalBorrowsNew$
- Emit a **Borrow** event with $borrower$, $borrowAmount$, $accountBorrowsNew$, $totalBorrowsNew$
- $call\ comptroller.borrowVerify(this,\ borrower,\ borrowAmount)$

[CErc20] Repay Borrow(uint repayAmount)

- Check $invoke\ Accrue\ Interest() = 0$
- return $invoke\ Repay\ Borrow\ Fresh(msg.sender,\ msg.sender,\ repayAmount)$

[CEther] Repay Borrow() payable

- Check $invoke\ Accrue\ Interest() = 0$
- return $invoke\ Repay\ Borrow\ Fresh(msg.sender,\ msg.sender,\ msg.value)$

[CErc20] Repay Borrow Behalf(address borrower, uint repayAmount)

Repays a borrow on behalf of another user. The message sender is still the payer, but you can specify a different account to pay against.

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Repay\ Borrow\ Fresh(msg.sender,\ borrower,\ repayAmount)$

[CEther] Repay Borrow Behalf(address borrower) payable

Repays a borrow on behalf of another user. The message sender is still the payer, but you can specify a different account to pay against.

- Check $invoke\ Accrue\ Interest() = 0$
- Return $invoke\ Repay\ Borrow\ Fresh(msg.sender,\ borrower,\ msg.value)$

[Internal] Repay Borrow Fresh(address payer, address borrower, uint repayAmount)

Borrows are repaid by the *payer* (possibly the same as the *borrower*).

- Fail if $call\ comptroller.repayBorrowAllowed(this, payer, borrower, repayAmount) \neq 0$
- Verify market's *block number* equals current block number
- We fetch the amount the *borrower* owes, with accumulated interest:
    - $accountBorrows = invoke\ Borrow\ Balance\ Stored(borrower)$
- If $repayAmount = -1$
    - $repayAmount = accountBorrows$
- Fail if *checkTransferIn(underlying, payer, repayAmount)* fails
- We calculate the new borrower and total borrow balances:
    - $accountBorrowsNew = accountBorrows - repayAmount$
        - Fails if $repayAmount > accountBorrows$
    - $totalBorrowsNew = totalBorrows - repayAmount$
        - Fails if $repayAmount > totalBorrows$
- **We have finished calculations**. (If any calculations failed with an error, we have already returned with a failure code). Now we can begin effects.
- We call *doTransferIn* for the *payer* and the *repayAmount*
    - Note: The cToken must handle variations between ERC-20 and ETH underlying
    - On success, the cToken holds an additional *repayAmount* of cash
    - If *doTransferIn* fails despite the fact we checked pre-conditions, we revert because we can't be sure if side effects occurred
- We write the previously calculated values into storage:
    - Set $accountBorrows_{borrower} = \{accountBorrowsNew,\ borrowIndex\}$
    - Set $totalBorrows = totalBorrowsNew$
- Emit **RepayBorrow** event with $payer$, $borrower$, $repayAmount$, $accountBorrowsNew$, $totalBorrowsNew$
- $call\ comptroller.repayBorrowVerify(this,\ payer,\ borrower,\ repayAmount)$

[CErc20] Liquidate Borrow(address borrower, CToken cTokenCollateral, uint repayAmount)

- Check $invoke\ Accrue\ Interest() = 0$
- Check $call\ cTokenCollateral.Accrue\ Interest() = 0$
- return $invoke\ Liquidate\ Borrow\ Fresh(msg.sender,\ borrower,\ repayAmount,\ cTokenCollateral)$

[CEther] Liquidate Borrow(address borrower, CToken cTokenCollateral) payable

- Check $invoke\ Accrue\ Interest() = 0$

- Check $call\ cTokenCollateral.Accrue\ Interest() = 0$
- return $invoke\ Liquidate\ Borrow\ Fresh(msg.sender,\ borrower,\ msg.value,\ cTokenCollateral)$

[Internal] Liquidate Borrow Fresh(address liquidator, address borrower, uint repayAmount, CToken cTokenCollateral)

The liquidator repays an amount of the underlying asset in this market, on behalf of an underwater borrower, and seizes the appropriate number of tokens in the collateral market.

- Fail if $call\ comptroller.liquidateBorrowAllowed(this, ...arguments) \neq 0$
- Verify market's *block number* equals current block number
- Verify *cTokenCollateral* market's *block number* equals current block number
  - Fail if $call\ cTokenCollateral.accrualBlockNumber() \neq block.number$
- Fail if $liquidator = borrower$
- Fail if $repayAmount = 0$
- Fail if $repayAmount = -1$
- We calculate the number of collateral tokens that will be seized:
  - $seizeTokens = call\ comptroller.liquidateCalculateSeizeTokens$ $(this,\ cTokenCollateral,\ repayAmount)$
- Fail if $seizeTokens > cTokenCollateral.balanceOf(borrower)$
- Fail if $invoke\ Repay\ Borrow\ Fresh(liquidator,\ borrower,\ repayAmount) \neq 0$
- Revert if $call\ cTokenCollateral.seize(liquidator,\ borrower,\ seizeTokens) \neq 0$
- Emit a **LiquidateBorrow** event with $liquidator,\ borrower$, $repayAmount$, $cTokenCollateral$, $seizeTokens$
- $call\ comptroller.liquidateBorrowVerify(this,\ ...arguments,\ ...state)$

seize(address liquidator, address borrower, uint seizeTokens) returns uint

- Fail if $call\ comptroller.seizeAllowed(this,\ msg.sender, liquidator, borrower, seizeTokens) \neq 0$
  - Note: It’s critical that the collateral contract uses *msg.sender* as the address of the borrowed CToken which it verifies with the Comptroller. If a parameter were used, then anyone would be able to spoof this call.
- Fail if $borrower = liquidator$
- We calculate the new borrower and liquidator token balances:
  - $borrowerTokensNew = accountTokens[borrower] - seizeTokens$
    - Fail on underflow
  - $liquidatorTokensNew = accountTokens[liquidator] + seizeTokens$
    - Fail on overflow
- We write the previously calculated values into storage:

    - $accountTokens[borrower] = borrowerTokensNew$
    - $accountTokens[liquidator] = liquidatorTokensNew$
- Emit a **Transfer** event
- $call\ comptroller.seizeVerify(this,\ msg.sender, liquidator, borrower,\ seizeTokens)$

Administrative Functions

constructor(EIP20 underlying, address interestRateModel, address comptroller, scaled initialExchangeRate)

- Set *admin* to msg.sender
- Set underlying to *underlying*
- Set initialExchangeRate to *initialExchangeRate*
- Set marketBlockNumber to block number
- Set market borrow index to 1e18
- Set reserve factor to 0
- $invoke\ \_setMarketInterestRateModelFresh(interestRateModel)$
- $invoke\ \_setMarketComptroller(comptroller)$

_setReserveFactor(scaled newReserveFactor)

- Check $invoke\ Accrue\ Interest() = 0$
- Return *_setReserveFactorFresh(newReserveFactor)*

[Internal] _setReserveFactorFresh(scaled newReserveFactor)

- Check caller is *admin*
- We verify market's *block number* equals current block number
- Check $newReserveFactor \leq maxReserveFactor$
- Store *reserveFactor* with value *newReserveFactor*
- Emit **NewReserveFactor***(oldReserveFactor, newReserveFactor)*

_reduceReserves(uint amount)

- Check $invoke\ Accrue\ Interest() = 0$
- Return *_reduceReservesFresh(amount)*

[Internal] _reduceReservesFresh(uint reduceAmount)

- Check caller is *admin*
- We verify market's *block number* equals current block number
- Check *amount* ≤ $reserves_n$

- Fail gracefully if protocol has insufficient underlying cash
- Store $reserves_{n+1} = reserves_n - reduceAmount$
- $invoke\ doTransferOut(underlying,\ reduceAmount,\ admin)$
    - Note: we revert on the failure of this command
- Emit **NewReserves**(*admin*, $reduceAmount$, $reserves_{n+1}$)

_setPendingAdmin(address newPendingAdmin)

- Check caller is *admin*
- Store *pendingAdmin* with value *newPendingAdmin*
- Emit **NewPendingAdmin***(oldPendingAdmin, newPendingAdmin)*

_acceptAdmin()

- Check caller is *pendingAdmin* and *pendingAdmin* ≠ address(0)
- Store *admin* with value *pendingAdmin*
- Unset *pendingAdmin*
- Emit **NewAdmin***(oldAdmin, newAdmin)*
- Emit **NewPendingAdmin***(oldPendingAdmin, 0)*

_setInterestRateModel(address newInterestRateModel)

- Check $invoke\ Accrue\ Interest() = 0$
- return *_setFreshInterestModelFresh(newInterestRateModel)*

[Internal] _setInterestRateModelFresh(address newInterestRateModel)

- Check caller is *admin*
- We assert market's *block number* equals current block number
- Track the market's current interest rate model
- Ensure $call\ newInterestRateModel.isInterestRateModel()$ returns true
- Set the interest rate model to *newInterestRateModel*
- Emit **NewInterestRateModel**(*oldInterestRateModel*, *newInterestRateModel*)

_setComptroller(address newComptroller)

- Check caller is *admin*
- Track asset's current comptroller as *oldComptroller*
- Ensure $call\ comptroller.isComptroller()$ returns true
- Set comptroller to *newComptroller*
- Emit **NewComptroller**(*oldComptroller*, *newComptroller*)

Token Functions

name() returns string

- Return *name*

symbol() returns string

- Return *symbol*

decimals() returns uint

- Return *decimals*

[External] getCash() returns uint

- Return $call\ getCashPrior()$

transfer(address **to,** uint **amount) returns** bool

- $invoke\ transferTokens(msg.sender,\ msg.sender,\ to,\ amount)$
    - Revert if not successful
    - Emit **Transfer***(msg.sender, to, tokens)*

transferFrom(address **from,** address **to,** uint **amount) returns** bool

- $invoke\ transferTokens(msg.sender,\ from,\ to,\ amount)$
    - Revert if not successful
    - Emit **Transfer***(from, to, tokens)*

[Internal] transferTokens(address **spender,** address **src,** address **dst,** uint **amount) returns** uint

Transfer *amount* cTokens from *source* to *dest.*

- Fail if $call\ comptroller.redeemAllowed(this,\ src,\ amount)$ returns false
- Fail unless *spender* is *source* or $cTokenAllowed[src][spender] \geq amount$
- Fail if $accountTokens[src] < amount$
- $accountTokens[src] -= amount$
    - Underflow is impossible due to check above
- $accountTokens[dst] += amount$
    - Fail on overflow
- Unless $spender = src\ \vee cTokenAllowed[src][spender] = maxUint$
    - $cTokenAllowed[src][spender] -= amount$

totalSupply() returns uint

- Return $totalSupply$

allowance(address owner, address spender) returns uint

- Return $cTokenAllowed[owner][spender]$

balanceOf(address account) returns uint256

- Return $accountTokens[account]$

balanceOfUnderlying(address account) returns uint

- Return $accountTokens[account] \times invoke\ Exchange\ Rate\ Current()$

approve(address spender, uint256 amount) returns bool

- Overwrite $cTokenAllowed[msg.sender][spender] = amount$
- Emit **Approval***(msg.sender, spender, amount)*

Exchange Rates

Exchange Rate Current() returns uint

- $invoke\ Accrue\ Interest()$
- return $invoke\ Exchange\ Rate\ Stored()$

Exchange Rate Stored()

- Note: we do not assert that the market is up to date.
- If there are no tokens minted:
    - $exchangeRate = initial\ exchange\ rate$
- Otherwise:
    - $totalCash = invoke\ getCash()$
        - *Note: likely makes an external call*
    - $exchangeRate = \frac{totalCash + totalBorrows - totalReserves}{totalSupply}$
- Return $exchangeRate$

Borrow Balances

Total Borrows Current(address account) returns uint

- $invoke\ Accrue\ Interest()$

- return $totalBorrows$

Borrow Balance Current(address account) returns uint

- $invoke\ Accrue\ Interest()$
- return $invoke\ Borrow\ Balance\ Stored()$

Borrow Balance Stored(address borrower)

- Note: we do not assert that the market is up to date.
- We get from storage from the cToken:
    - $borrowBalance_{borrower} = accountBorrows[borrower]$
    - $borrowIndex_{borrower} = accountBorrowIndex[borrower]$
- If $borrowBalance_{borrower} = 0$ then $borrowIndex_{borrower}$ is likely also 0. Rather than failing the calculation with a division by 0, we immediately return 0 in this case.
- $recentBorrowBalance_{borrower} = \frac{borrowBalance_{borrower} \times borrowIndex_{stored}}{borrowIndex_{borrower}}$
- Return $recentBorrowBalance_{borrower}$

Safe Token

[Internal] checkTransferIn(address account, uint underylingAmount) returns uint

- $call\ EIP20(underlying).allowance(account,\ address(this))$
    - Fail if result is less than *underlyingAmount*
- $call\ EIP20(underlying).balanceOf(account)$
    - Fail if result is less than *underlyingAmount*
- Return 👍

[Internal] doTransferIn(address account, uint underlyingAmount)

- Revert if $msg.value > 0$ since there is no valid use case for sending value by default
- $call\ EIP20(underlying).transferFrom(account,\ address(this),\ underlyingAmount)$
    - Revert unless true
    - *Note: should handle non-standard ERC-20 tokens

[Internal] getCashPrior() returns uint

- Return $call\ EIP20(underlying).balanceOf(address(this))$

[Internal] doTransferOut(address account, uint underlyingAmount)

- $call\ EIP20(underlying).transfer(account,\ underlyingAmount)$
    - Revert unless true

    - *Note: should handle non-standard ERC-20 tokens

Safe Token (cETH)

In order to implement cETH, we add a fallback function that does $invoke\ mint(msg.value)$.
In addition, we override the Safe Token methods as follows:

[Internal] **checkTransferIn(**address **account,** uint **underylingAmount) returns** uint

- Fail if $msg.sender \neq account$
- Fail if $msg.value \neq underlyingAmount$
- Return👍

[Internal] **doTransferIn(**address **account,** uint **underlyingAmount)**

- Fail if $msg.value \neq underlyingAmount$
    - We just sanity check, *checkTransferIn* should have already been called

[Internal] **getCashPrior() returns** uint

- Return $this.balance - msg.value$
    - Ensure we avoid underflow

[Internal] **doTransferOut(**address **account,** uint **underlyingAmount)**

- $invoke\ account.transfer(underlyingAmount)$
    - Ensure minimum gas is attached to transfer

Comptroller Contract

The Comptroller implements the Liquidity Checker API specification. Most important of these are $liquidityChecker$. *redeemAllowed(),* $liquidityChecker$. *borrowAllowed(),* and $liquidityChecker$. *liquidateBorrowAllowed()*.

The Comptroller also implements a defense hook mechanism to protect against unforeseen future vulnerabilities. These **Verify* functions are currently no-ops, but provide a last resort to potentially revert any protocol transaction which would violate the intended behavior of the protocol and therefore put user assets at risk.

Note: In order to seamlessly upgrade the Comptroller without changing the Comptroller address referenced by the cToken markets, we sometimes use a technique known as *delegate calls*.

User Market Functions

These two functions (enter markets and exit market) will be called by the end-users directly. This will only be a requirement for users who wish to borrow. That is, token holders that do not borrow will not need to (and should not) call these functions.

Enter Markets(address[] cTokens) returns bool[]

The sender includes a list of asset addresses that should be used when calculating account liquidity. Before borrowing an asset, one or more supplied assets must be added in this way to provide collateral. Any asset to be borrowed must be added in this way before borrowing is allowed. The return value is a list of mapping the assets passed in to whether the user is ultimately in that market.

- For each cToken given as an argument:
  - We check if the user is already in the cToken, if so, collect true, otherwise, proceed
  - We check if the user has reached maxAssets, if so, collect false, otherwise, proceed
  - We check if the cToken is listed, if not, collect false, otherwise, proceed.
  - We check $oraclePrice_{asset} \neq 0$
  - We add the asset to assets$_{user}$, by pushing it into user's assets list and setting $memberships_{cToken,user}$ to true, and collect true
  - Proceed to the next asset, noting that the size of the list was increased if the previous item added to the list and the maxAssets comparison must occur against storage
  - Emit **MarketEntered**(*cToken, msg.sender*)
- Return the collected answers of whether the user is currently in the passed cTokens

Exit Market(address cToken) returns bool

The sender provides an asset that they no longer wish to be included in account liquidity calculations. Since all borrowed assets **must** be included, the purpose of the function is to remove an asset from the user's collateral list. The return value is a boolean indicating if the user is not in the market after the call.

- Get sender *tokensHeld* and *amountOwed* underlying from the cToken
- Fail if the sender has a borrow balance
  - i.e. $amountOwed \neq 0$
- Fail if the sender is not permitted to redeem all of their tokens

    - i.e. $invoke\ redeemAllowed(cToken,\ msg.sender,\ tokensHeld) \neq 0$
- Return true if the sender is not already 'in' the market
- Set *cToken* account membership to false
- Delete *cToken* from the account's list of assets
- Emit **MarketExited**(*cToken, msg.sender*)
- Return true, indicating the user is no longer in the market

Policy Hook Functions

These functions are core to verifying if a given action by a user is allowed. This should be based on a combination of factors. One or more of the factors will be checked by the following functions.

1. cToken must be a known "supported" asset. **This applies to all functions and must always be checked.**
2. User must remain sufficiently liquid after the function were to complete. For instance, a user cannot redeem tokens if she has too many outstanding borrows.
3. User must declare all assets she intends to borrow (and use as collateral).
4. User must pass all KYC-type checks and other policy rules.
5. For liquidation, caller must be the asset itself (the borrowed cToken contract).

mintAllowed(CToken **cToken,** address **minter,** uint **mintAmount) returns** uint

- Fail if *cToken* not listed
- **may include Policy Hook-type checks*
- Return 👍 otherwise

mintVerify(CToken **cToken,** address **minter,** uint **mintAmount,** uint **mintTokens) returns** uint

- Does nothing, but could revert in the future as a defense hook

redeemAllowed(CToken **cToken,** address **redeemer,** uint **redeemTokens) returns** uint

- Fail if *cToken* not listed
- **may include Policy Hook-type checks*
- Return 👍 if *redeemer* does not have membership in asset
- Let $(error,\ liquidity,\ shortfall)\ =$ $invoke\ getHypotheticalAccountLiquidityInternal(redeemer,\ cToken,\ redeemTokens,\ 0)$
- Fail if $error \neq 0$ or if $shortfall > 0$
- Return 👍 otherwise

redeemVerify(CToken **cToken,** address **redeemer,** uint **redeemAmount,** uint **redeemTokens) returns** uint

- Does nothing, but could revert in the future as a defense hook

borrowAllowed(CToken **cToken,** address **borrower,** uint **borrowAmount) returns** uint

- Fail *if cToken not listed*
- **may include Policy Hook-type checks*
- Fail if *borrower* does not have membership in asset
- Fail if $oraclePrice_{asset} = 0$
- Let $(error, liquidity, shortfall) = invoke\ getHypotheticalAccountLiquidityInternal(borrower, cToken, 0, borrowAmount)$
- Fail if $error \neq 0$ or $shortfall > 0$
- Return 👍 otherwise

borrowVerify(CToken **cToken,** address **borrower,** uint **borrowAmount)**

- Does nothing, but could revert in the future as a defense hook

repayBorrowAllowed(CToken **cToken,** address **payer,** address **borrower,** uint **repayAmount) returns** uint

- Fail if *cToken* not listed
- **may include Policy Hook-type checks*
- Return 👍 otherwise

repayBorrowVerify(CToken **cToken,** address **payer,** address **borrower,** uint **repayAmount)**

- Does nothing, but could revert in the future as a defense hook

liquidateBorrowAllowed(CToken **cTokenBorrowed,** CToken **cTokenCollateral,** address **borrower,** address **liquidator,** uint **repayAmount) returns** uint

- Fail if *cTokenBorrowed* or *cTokenCollateral* not listed
- **may include Policy Hook-type checks*
- Let $(error, liquidity, shortfall) = invoke\ getAccountLiquidityInternal(borrower)$
- Fail if $error \neq 0$ or $shortfall = 0$
    - The borrower must have shortfall in order to be liquidatable
- $borrowBalance_{account} = call\ cTokenBorrowed.Borrow\ Balance\ Stored()$
    - This value is strictly up-to-date due to accumulating interest prior to this call
- We calculate *maxCloseValue*, the total that can be closed for this borrow:
    - $maxCloseValue = borrowBalance_{account} \cdot closeFactor$

- Fail if $repayAmount > maxCloseValue$
- Return 👍 otherwise

liquidateBorrowVerify(CToken **cTokenBorrowed,** CToken **cTokenCollateral,** address **borrower,** address **liquidator,** uint **repayAmount,** uint **seizeTokens)**

- Does nothing, but could revert in the future as a defense hook

seizeAllowed(CToken **cTokenCollateral,** CToken **cTokenBorrowed,** address **borrower,** address **liquidator,** uint **seizeTokens) returns** uint

- Fail if *cTokenCollateral* or *cTokenBorrowed* is not listed
- **may include Policy Hook-type checks*
- Fail $call\ cTokenCollateral.comptroller() \neq call\ cTokenBorrowed.comptroller()$
- Return 👍 otherwise

seizeVerify(CToken **cTokenCollateral,** CToken **cTokenBorrowed,** address **borrower,** address **liquidator,** uint **seizeTokens)**

- Does nothing, but could revert in the future as a defense hook

transferAllowed(CToken **cToken,** address **src,** address **dst,** uint **transferTokens) returns** uint

- Return $redeemAllowed(cToken,\ src,\ transferTokens)$

transferVerify(CToken **cToken,** address **src,** address **dst,** uint **transferTokens)**

- Does nothing, but could revert in the future as a defense hook

Liquidity / Liquidation Calculations

getAssetsIn(address **account) returns** address[]

- Return list of assets you are in

checkMembership(address **account,** CToken **cToken) returns** bool

- Returns true if user is in asset

getAccountLiquidity(address **account) returns** int

- Return $invoke\ getHypotheticalAccountLiquidity(account,\ CToken(0),\ 0,\ 0)$

getHypotheticalAccountLiquidity(address **account,** CToken **cToken,** uint **redeemTokens,** uint **borrowAmount) returns (uint, uint)**

- Let $assets_{account}$ be the active list of assets (from storage) that a user has entered
- We calculate the user's $sumCollateral$ and $sumBorrowPlusEffects$
  - Note that we calculate the *exchangeRateStored* for each collateral cToken using stored data, without calculating accumulated interest
- Initialize $sumCollatera = 0$, $sumBorrowPlusEffects = 0$
- For each $asset \in assets_{account}$:
  - We get:
    - $cTokenBalance_{account} = call\ cToken.balanceOf(account)$
    - $borrowBalance_{account} = call\ cToken.Borrow\ Balance\ Stored(account)$
    - $exchangeRate = call\ cToken.Exchange\ Rate\ Stored()$
    - $collateralFactor = markets[asset].collateralFactor$
    - $oraclePrice = call\ oracle.getUnderlyingPrice(asset)$
      - Fail if $oraclePrice = 0$
  - $tokensToDollars = collateralFactor \cdot exchangeRate \cdot oraclePrice$
  - $sumCollateral \mathrel{+}= tokensToDollars \cdot cTokenBalance_{account}$
  - $sumBorrowPlusEffects \mathrel{+}= oraclePrice \cdot borrowBalance_{account}$
  - If $asset = cToken$ (i.e. looking at the affected market):
    - Account for the potential effect of redeeming:
      - $sumBorrowPlusEffects \mathrel{+}= tokensToDollars \cdot redeemTokens$
    - Account for the potential effect of borrowing:
      - $sumBorrowPlusEffects \mathrel{+}= oraclePrice \cdot borrowAmount$
- If $sumCollateral > sumBorrowPlusEffects$
  - $liquidity = sumCollateral - sumBorrowPlusEffects$
  - $shortfall = 0$
- Else
  - $liquidity = 0$
  - $shortfall = sumBorrowPlusEffects - sumCollateral$
- Return two unsigned values, $liquidity$ and $shortfall$

liquidateCalculateSeizeTokens(CToken **cTokenBorrowed,** CToken **cTokenCollateral,** uint **repayAmount) returns** uint

- Read oracle prices for borrowed and collateral markets:
  - $price_{borrowed} = call\ oracle.getUnderlyingPrice(cTokenBorrowed)$
  - $price_{collateral} = call\ oracle.getUnderlyingPrice(cTokenCollateral)$

    - Fail if either $price_{borrowed} = 0$ or $price_{collateral} = 0$
- Get the exchange rate and calculate the number of collateral tokens to seize:
    - $exchangeRate_{collateral} = call\ cTokenCollateral.Exchange\ Rate\ Stored()$
    - $seizeAmount\ =\ repayAmount \times liquidationIncentive_{stored} \times \frac{price_{borrowed}}{price_{collateral}}$
    - $seizeTokens\ =\ seizeAmount \div exchangeRate_{collateral}$
- Return $seizeTokens$

Comptroller Admin Functions

constructor()

- Set admin to caller

_setPendingAdmin(address newPendingAdmin) returns uint

- Check caller is admin
- Store pendingAdmin with value newPendingAdmin
- Emit **NewPendingAdmin**(oldPendingAdmin, newPendingAdmin)

_acceptAdmin() returns uint

- Check caller is pendingAdmin and pendingAdmin ≠ address(0)
- Store admin with value pendingAdmin
- Unset pendingAdmin
- Emit **NewAdmin***(oldAdmin, newAdmin)*
- Emit **NewPendingAdmin***(oldPendingAdmin, 0)*

_setPriceOracle(address newOracle)

- Check caller is *admin* OR caller is *currentImplementation* and origin is *admin*
- Ensure $invoke\ newOracle.isPriceOracle()$ returns true
- Set the comptroller's oracle to *newOracle*
- Emit **NewPriceOracle**(*oldOracle*, *newOracle*)

_setLiquidationIncentive(scaled newIncentive)

- Check caller is *admin* OR caller is *currentImplementation* and origin is *admin*
- Check de-scaled 1 ≤ *newLiquidationDiscount* ≤ 1.5
- Set liquidation incentive to *newIncentive*
- Emit **NewLiquidationIncentive**(oldIncentive, newIncentive)

_setCollateralFactor(CToken cToken, scaled newFactor) returns uint

- Check caller is *admin* OR caller is *currentImplementation* and origin is *admin*
- Verify market is listed
- Check *newFactor* $\leq$ 0.9
- If *newFactor* > 0, fail if $oracle.getUnderlyingPrice(cToken) \neq 0$
- Set market's collateral factor to *newFactor*
- Emit **NewCollateralFactor**(*cToken*, *oldFactor*, *newFactor)*

_setCloseFactor(scaled newCloseFactor) returns uint

- Check caller is *admin* OR caller is *currentImplementation* and origin is *admin*
- Check 0.05 $<$ *newCloseFactor* $\leq$ 0.9
- Set close factor to *newCloseFactor*
- Emit **NewCloseFactor**(old*CloseFactor*, *newCloseFactor)*

_setMaxAssets(uint newMaxAssets) returns uint

- Check caller is *admin* OR caller is *currentImplementation* and origin is *admin*
- Set maxAssets to *newMaxAssets*
- Emit **NewMaxAssets***(oldMaxAssets, newMaxAssets)*

_supportMarket(CToken cToken) returns uint

- Check caller is admin
- Verify asset is not isListed
- Ensure $call\ cToken.isCToken()$ returns true
- Set market is listed to true
- Append cToken to *markets* list.
- Emit **MarketListed**(cToken)

Implementation Upgrade Functions

The comptroller is designed as an upgradeable proxy inspired by patterns described by zeppelinOS. In short, the pattern is

1. Deploy new implementation
2. $call\ comptroller.\_setPendingImplementation(newImplementation)$
3. $call\ newImplementation.becomeBrains(comptroller,\ ...)$

_setPendingImplementation(address newPendingImplementation) returns uint

- Check caller is admin

- Store *pendingComptrollerImplementation* with value *newPendingImplementation*
- Emit **NewPendingImplementation**(*oldPendingAdminImplementation*, *newPendingImplementation*)

_acceptImplementation() returns uint

- Check *caller* is *pendingComptrollerImplementation* and *pendingComptrollerImplementation ≠ address(0)*
- Store *comptrollerImplementation* with value *pendingComptrollerImplementation*
- Unset *pendingComptrollerImplementation*
- Emit **NewImplementation***(oldImplementation, newImplementation)*
- Emit **NewPendingImplementation***(oldPendingImplementation, 0)*

Note: _becomeBrains is called on the implementation address, where the other functions are all called on the active comptroller.

_becomeBrains(address **unitroller,** address **oracle,** uint **closeFactor,** uint **maxAssets) returns** uint

- Check caller is admin of unitroller
- Ensure $call\ unitroller.\_acceptImplementation()$ returns true
- Ensure $call\ unitroller.\_setMarketPriceOracle(oracle)$ = 0
- Ensure $call\ unitroller.\_setCloseFactor(closeFactor) = 0$
- Ensure $call\ unitroller.\_setMaxAssets(maxAssets) = 0$
- Ensure $call\ unitroller.\_setLiquidationIncentive(liquidationIncentiveMinMantissa) = 0$

Maximillion Contract

For cErc20 contracts, we support repaying a borrow fully using the special '-1' amount. Since the CToken contract is approved to transfer what it needs, it can determine the borrow amount and then transfer the exact amount, post-interest accrual, directly within the repayBorrow function.

For cEther, things work a bit differently. Since '-1' is actually UINT_MAX, it's practically impossible for anyone to transfer this amount to the repay function. In order to completely repay a borrow in cEther, we deploy a separate contract to handle the details of collecting more than enough Ether to repay the borrow plus recent interest, and then refunding the overpay amount.

Repay Behalf(address borrower) payable

- Remember the amount of Ether received:
    - $received = msg.value$
- Read the current borrow balance with interest accrued:
    - $borrows = invoke\ cEther.borrowBalanceCurrent(borrower)$
- If $received > borrows$, repay the exact borrow balance and refund:
    - $invoke\ cEther.repayBorrowBehalf.value(borrows)(borrower)$
    - Refund $received - borrows$
- Otherwise, just repay the amount of Ether provided:
    - $invoke\ cEther.repayBorrowBehalf.value(received)(borrower)$

Appendix

Market States

A given asset may be in one of four states, which affect which functions are available and how the asset is utilized above.

- Unsupported - An asset is not part of the "listedAssets" set. It is not used when calculating sumCollateral and all operations, aside of "supportMarket" for that asset should fail.
- Listed - An asset is part of the "listedAssets" set. It is not used when calculating sumCollateral and only "supply", "withdraw" and "repayBorrow" operations on that asset should functional normally. The asset must have an interest rate model associated with it.
- Borrow - An asset is part of the "listedAssets" set. It is not used when calculating sumCollateral and all operations on that asset should functional normally. The asset must have a non-zero price and interest rate model associated with it. A listed market with non-zero price allows that market to be borrowed from.
- Collateral - An asset is part of the "listedAssets" set. It is used when calculating sumCollateral and all operations on that asset should functional normally. The asset must have a non-zero price, non-zero collateral factor and interest rate model associated with it. A borrow market with a non-zero collateral factor is both borrowable and can be used as collateral.

Regarding membership, $listed \subseteq borrow \subseteq collateral$ and $(listed \cap unsupported) = \emptyset$
Regarding available functionality, $listed \supseteq borrow \supseteq collateral$ and $unsupported = \emptyset$

Interest Index Calculation

The interest index tracks the interest owed on $1 (or some constant initial amount) of debt since the protocol's deployment. That is, the interest on a fixed amount of the borrowed asset over time.

Whenever the interest rate changes, the index applies the simple interest formula, to snapshot the effect of the prior interest rate since that time:

$$\begin{aligned} index_0 &= c,\ c > 0 \\ index_{i+\Delta blocks} &= index_i \cdot (1 + \Delta blocks \times rate_i) \end{aligned}$$

Whenever the index is updated, we also update the total borrow (and reserves) amount, to capture the effect on all the currently borrowed (reserved) units of the asset since the last update:

$$borrows_{i+\Delta blocks} = borrows_i \cdot (1 + \Delta blocks \times rate_i)$$

The total borrow amount is used to manage the market's ledger, and is updated often to maintain accurate current information. On the other hand, each individual borrow is only updated when an action related to that specific borrow is taken.

When a borrow is created, we store with it the principal amount and the interest index at that time. Whenever an action which affects the principal of the borrow is initiated, we first calculate a new principal, based on the interest that has accumulated since the last event.

In order to determine how much interest has accumulated, we take the current index value and compare it to the interest index at the time of the last event which was stored in the borrow balance. The ratio of these values yields the change in $1 of debt since the principal was last recorded, for which we then calculate a new principal amount to store with the new index.

The ratio $borrowIndex_{i+b}/borrowIndex_i$ is the simple interest rate over the period, which we apply for the whole period in order to accrue interest on the principal:

$$\begin{aligned} effectiveRate &= borrowIndex_{i+\Delta blocks} \,/\, borrowIndex_i \\ principal_{i+\Delta blocks} &= principal_i \cdot (1 + \Delta blocks \cdot effectiveRate) + \Delta \end{aligned}$$

$$borrowIndex_{i+\triangle blocks} = borrowIndex_i$$

Mint / Redeem exchange rate remains the same when minting and redeeming coins

Let

$A \ = \ assets \ = \ cash \ + \ borrows\ held\ by\ the\ protocol$

$L \ = \ liabilities \ = \ tokens\ minted\ and\ not\ redeemed$

$dA \ = \ change\ in\ assets \ = \ cash\ supplied|withdrawn\ or\ borrows\ sent|repaid$

$dL \ = \ change\ in\ liabilities \ = \ tokens\ to\ mint|redeem$

$R \ = \ exchange\ rate \ = \ A\ /\ L$

$A' \ = \ next\ assets \ = \ A \ + \ dA$

$L' \ = \ next\ liabilities \ = \ L \ + \ dL$

$R' \ = \ next\ exchange\ rate \ = \ A'\ /\ L'$

Prove that $R' \ = \ R$, whenever there is a change in dA, where $dA\ /\ dL \ = \ R \ = \ A\ /\ L$.

$dL \ = \ dA\ /\ R = \ L \ * \ dA\ /\ A$

$$\begin{aligned}
R' &= A'\ /\ L' \\
&= (A \ + \ dA)\ /\ (L \ + \ dL) \\
&= (A \ + \ dA)\ /\ (L \ + \ L \ * \ dA\ /\ A) \\
&= (A \ + \ dA)\ /\ (L \ * \ (1 \ + \ dA\ /\ A)) \\
&= (A \ + \ dA)\ /\ (L \ * \ (A\ /\ A \ + \ dA\ /\ A)) \\
&= (A \ + \ dA)\ /\ (L \ * \ (A \ + \ dA)\ /\ A) \\
&= 1\ /\ L\ /\ A \\
&= A\ /\ L \\
&= R
\end{aligned}$$

q.e.d.