Markets
Governance
Docs

- Compound II
- cTokens
- Comptroller
- Governance
- Open Price Feed
- Security

Compound v2

cTokens
Comptroller
Governance
Open Price Feed
Security

Comptroller

Architecture

Enter Markets

Exit Market

Get Assets In

Collateral Factor

Get Account Liquidity

Close Factor

Liquidation Incentive

Key Events

Error Codes

COMP Distribution Speeds

Claim COMP

Market Metadata

Comptroller

Introduction

The Comptroller is the risk management layer of the Compound protocol; it determines how much collateral a user is required to maintain, and whether (and by how much) a user can be liquidated. Each time a user interacts with a cToken, the Comptroller is asked to approve or deny the transaction.

The Comptroller maps user balances to prices (via the Price Oracle) to risk weights (called Collateral Factors) to make its determinations. Users explicitly list which assets they would like included in their risk scoring, by calling Enter Markets and Exit Market.

Architecture

The Comptroller is implemented as an upgradeable proxy. The Unitroller proxies all logic to the Comptroller implementation, but storage values are set on the Unitroller. To call Comptroller functions, use the Comptroller ABI on the Unitroller address.

Enter Markets

Enter into a list of markets - it is not an error to enter the same market more than once. In order to supply collateral or borrow in a market, it must be entered first.

Comptroller

```
function enterMarkets(address[] calldata cTokens)
returns (uint[] memory)
```

- `msg.sender`: The account which shall enter the given markets.
- `cTokens`: The addresses of the cToken markets to enter.

- `RETURN`: For each market, returns an error code indicating whether or not it was entered. Each is 0 on success, otherwise an Error code.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

CToken[] memory cTokens = new CToken[](2);
cTokens[0] = CErc20(0x3FDA...);
cTokens[1] = CEther(0x3FDB...);
uint[] memory errors = troll.enterMarkets(cTokens);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const cTokens = [CErc20.at(0x3FDA...),
CEther.at(0x3FDB...)];
const errors = await
troll.methods.enterMarkets(cTokens).send({from: ...});
```

Exit Market

Exit a market - it is not an error to exit a market which is not currently entered. Exited markets will not count towards account liquidity calculations.

Comptroller

```
function exitMarket(address cToken) returns (uint)
```

- `msg.sender`: The account which shall exit the given market.
- `cTokens`: The addresses of the cToken market to exit.

- `RETURN`: 0 on success, otherwise an Error code.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

uint error = troll.exitMarket(CToken(0x3FDA...));
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const errors = await
troll.methods.exitMarket(CEther.at(0x3FDB...)).send({from: ...});
```

Get Assets In

Get the list of markets an account is currently entered into. In order to supply collateral or borrow in a market, it must be entered first. Entered markets count towards account liquidity calculations.

Comptroller

```
function getAssetsIn(address account) view returns
(address[] memory)
```

- `account`: The account whose list of entered markets shall be queried.
- `RETURN`: The address of each market which is currently entered into.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

address[] memory markets =
troll.getAssetsIn(0xMyAccount);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const markets = await
troll.methods.getAssetsIn(cTokens).call();
```

Collateral Factor

A cToken’s collateral factor can range from 0-90%, and represents the proportionate increase in liquidity (borrow limit) that an account receives by minting the cToken. Generally, large or liquid assets have high collateral factors, while small or illiquid assets have low collateral factors. If an asset has a 0% collateral factor, it can’t be used as collateral (or seized in liquidation), though it can still be borrowed.

> Collateral factors can be increased (or decreased) through Compound Governance, as market conditions change.

Comptroller

```
function markets(address cTokenAddress) view returns
(bool, uint, bool)
```

- `cTokenAddress`: The address of the cToken to check if listed and get the collateral factor for.
- `RETURN`: Tuple of values (isListed, collateralFactorMantissa, isComped); isListed represents whether the comptroller recognizes this cToken; collateralFactorMantissa, scaled by 1e18, is multiplied by a supply balance to determine how much value can be borrowed. The isComped boolean indicates whether or not suppliers and borrowers are distributed COMP tokens.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

(bool isListed, uint collateralFactorMantissa, bool
isComped) = troll.markets(0x3FDA...);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const result = await
troll.methods.markets(0x3FDA...).call();
const {0: isListed, 1: collateralFactorMantissa, 2:
isComped} = result;
```

Get Account Liquidity

Account Liquidity represents the USD value borrowable by a user, before it reaches liquidation. Users with a shortfall (negative liquidity) are subject to liquidation, and can’t withdraw or borrow assets until Account Liquidity is positive again.

For each market the user has entered into, their supplied balance is multiplied by the market’s collateral factor, and summed; borrow balances are then subtracted, to equal Account Liquidity. Borrowing an asset reduces Account Liquidity for each USD borrowed; withdrawing an asset reduces Account Liquidity by the asset's collateral factor times each USD withdrawn.

Because the Compound Protocol exclusively uses unsigned integers, Account Liquidity returns either a surplus or shortfall.

Comptroller

```
function getAccountLiquidity(address account) view
returns (uint, uint, uint)
```

- `account`: The account whose liquidity shall be calculated.
- `RETURN`: Tuple of values (error, liquidity, shortfall). The error shall be 0 on success, otherwise an error code. A non-zero liquidity value indicates the account has available account liquidity. A non-zero shortfall value indicates the account is currently below his/her collateral requirement and is subject to liquidation. At most one of liquidity or shortfall shall be non-zero.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

(uint error, uint liquidity, uint shortfall) =
troll.getAccountLiquidity(msg.caller);
require(error == 0, "join the Discord");
require(shortfall == 0, "account underwater");
require(liquidity > 0, "account has excess
collateral");
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const result = await
troll.methods.getAccountLiquidity(0xBorrower).call();
const {0: error, 1: liquidity, 2: shortfall} = result;
```

Close Factor

The percent, ranging from 0% to 100%, of a liquidatable account's borrow that can be repaid in a single liquidate transaction. If a user has multiple borrowed assets, the closeFactor applies to any single borrowed asset, not the aggregated value of a user's outstanding borrowing.

Comptroller

```
function closeFactorMantissa() view returns (uint)
```

- `RETURN`: The closeFactor, scaled by 1e18, is multiplied by an outstanding borrow balance to determine how much could be closed.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

uint closeFactor = troll.closeFactorMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const closeFactor = await
troll.methods.closeFactorMantissa().call();
```

Liquidation Incentive

The additional collateral given to liquidators as an incentive to perform liquidation of underwater accounts. A portion of this is given to the collateral cToken reserves as determined by the seize share. The seize share is assumed to be 0 if the cToken does not have a `protocolSeizeShareMantissa` constant. For example, if the liquidation incentive is 1.08, and the collateral's seize share is 1.028, liquidators receive an extra 5.2% of the borrower's collateral for every unit they close, and the remaining 2.8% is added to the cToken's reserves.

Comptroller

```
function liquidationIncentiveMantissa() view returns
(uint)
```

- `RETURN`: The liquidationIncentive, scaled by 1e18, is multiplied by the closed borrow amount from the liquidator to determine how much collateral can be seized.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

uint closeFactor =
troll.liquidationIncentiveMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);

const closeFactor = await
troll.methods.liquidationIncentiveMantissa().call();
```

Key Events

Event	Description
`MarketEntered(CToken cToken, address account)`	Emitted upon a successful Enter Market.
`MarketExited(CToken cToken, address account)`	Emitted upon a successful Exit Market.

Error Codes

Code	Name	Description
0	`NO_ERROR`	Not a failure.
1	`UNAUTHORIZED`	The sender is not authorized to perform this action.
2	`COMPTROLLER_MISMATCH`	Liquidation cannot be performed in markets with different comptrollers.

Code	Name	Description
3	`INSUFFICIENT_SHORTFALL`	The account does not have sufficient shortfall to perform this action.
4	`INSUFFICIENT_LIQUIDITY`	The account does not have sufficient liquidity to perform this action.
5	`INVALID_CLOSE_FACTOR`	The close factor is not valid.
6	`INVALID_COLLATERAL_FACTOR`	The collateral factor is not valid.
7	`INVALID_LIQUIDATION_INCENTIVE`	The liquidation incentive is invalid.
8	`MARKET_NOT_ENTERED`	The market has not been entered by the account.
9	`MARKET_NOT_LISTED`	The market is not currently listed by the comptroller.
10	`MARKET_ALREADY_LISTED`	An admin tried to list the same market more than once.
11	`MATH_ERROR`	A math calculation error occurred.

Code	Name	Description
12	NONZERO_BORROW_BALANCE	The action cannot be performed since the account carries a borrow balance.
13	PRICE_ERROR	The comptroller could not obtain a required price of an asset.
14	REJECTION	The comptroller rejects the action requested by the market.
15	SNAPSHOT_ERROR	The comptroller could not get the account borrows and exchange rate from the market.
16	TOO_MANY_ASSETS	Attempted to enter more markets than are currently supported.
17	TOO_MUCH_REPAY	Attempted to repay more than is allowed by the protocol.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK
2	EXIT_MARKET_BALANCE_OWED
3	EXIT_MARKET_REJECTION
4	SET_CLOSE_FACTOR_OWNER_CHECK
5	SET_CLOSE_FACTOR_VALIDATION
6	SET_COLLATERAL_FACTOR_OWNER_CHECK
7	SET_COLLATERAL_FACTOR_NO_EXISTS
8	SET_COLLATERAL_FACTOR_VALIDATION
9	SET_COLLATERAL_FACTOR_WITHOUT_PRICE
10	SET_IMPLEMENTATION_OWNER_CHECK
11	SET_LIQUIDATION_INCENTIVE_OWNER_CHECK
12	SET_LIQUIDATION_INCENTIVE_VALIDATION
13	SET_MAX_ASSETS_OWNER_CHECK

Code	Name
14	SET_PENDING_ADMIN_OWNER_CHECK
15	SET_PENDING_IMPLEMENTATION_OWNER_CHECK
16	SET_PRICE_ORACLE_OWNER_CHECK
17	SUPPORT_MARKET_EXISTS
18	SUPPORT_MARKET_OWNER_CHECK

COMP Distribution Speeds

COMP Speed

The “COMP speed” unique to each market is an unsigned integer that specifies the amount of COMP that is distributed, per block, to suppliers and borrowers in each market. This number can be changed for individual markets by calling the `_setCompSpeed` method through a successful Compound Governance proposal. The following is the formula for calculating the rate that COMP is distributed to each supported market.

```
utility = cTokenTotalBorrows * assetPrice

utilityFraction = utility /
sumOfAllCOMPedMarketUtilities
marketCompSpeed = compRate * utilityFraction
```

COMP Distributed Per Block (All Markets)

The Comptroller contract’s `compRate` is an unsigned integer that indicates the rate at which the protocol distributes COMP to markets’ suppliers or borrowers, every Ethereum block. The value is the amount of COMP (in wei), per block, allocated for the markets. Note that not every market has COMP distributed to its participants (see Market Metadata). The compRate indicates how much COMP goes to the suppliers or borrowers, so doubling this number shows how much COMP goes to all suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to all markets.

Comptroller

```
uint public compRate;
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

// COMP issued per block to suppliers OR borrowers *
(1 * 10 ^ 18)
uint compRate = troll.compRate();
// Approximate COMP issued per day to suppliers OR
borrowers * (1 * 10 ^ 18)
uint compRatePerDay = compRate * 4 * 60 * 24;
// Approximate COMP issued per day to suppliers AND
borrowers * (1 * 10 ^ 18)
uint compRatePerDayTotal = compRatePerDay * 2;
```

Web3 1.2.6

```
const comptroller = new
web3.eth.Contract(comptrollerAbi, comptrollerAddress);

let compRate = await
comptroller.methods.compRate().call();
compRate = compRate / 1e18;
// COMP issued to suppliers OR borrowers
const compRatePerDay = compRate * 4 * 60 * 24;
// COMP issued to suppliers AND borrowers
const compRatePerDayTotal = compRatePerDay * 2;
```

COMP Distributed Per Block (Single Market)

The Comptroller contract has a mapping called `compSpeeds`. It maps cToken addresses to an integer of each market's COMP distribution per Ethereum block. The integer indicates the rate at which the protocol distributes COMP to markets' suppliers or borrowers. The value is the amount of COMP (in wei), per block, allocated for the market. Note that not every market has COMP distributed to its participants (see Market Metadata). The speed indicates how much COMP goes to the suppliers or the borrowers, so doubling this number shows how much COMP goes to market suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to a single market.

Comptroller

```
mapping(address => uint) public compSpeeds;
```

Solidity

```
Comptroller troll = Comptroller(0x123...);

address cToken = 0xabc...;
```

```
// COMP issued per block to suppliers OR borrowers *
(1 * 10 ^ 18)
uint compSpeed = troll.compSpeeds(cToken);
// Approximate COMP issued per day to suppliers OR
borrowers * (1 * 10 ^ 18)
uint compSpeedPerDay = compSpeed * 4 * 60 * 24;
// Approximate COMP issued per day to suppliers AND
borrowers * (1 * 10 ^ 18)
uint compSpeedPerDayTotal = compSpeedPerDay * 2;
```

Web3 1.2.6

```
const cTokenAddress = '0xabc...';

const comptroller = new
web3.eth.Contract(comptrollerAbi, comptrollerAddress);
let compSpeed = await
comptroller.methods.compSpeeds(cTokenAddress).call();
compSpeed = compSpeed / 1e18;
// COMP issued to suppliers OR borrowers
const compSpeedPerDay = compSpeed * 4 * 60 * 24;
// COMP issued to suppliers AND borrowers
const compSpeedPerDayTotal = compSpeedPerDay * 2;
```

Claim COMP

Every Compound user accrues COMP for each block they are supplying to or borrowing from the protocol. Users may call the Comptroller’s `claimComp` method at any time to transfer COMP accrued to their address.

Comptroller

```
// Claim all the COMP accrued by holder in all markets

function claimComp(address holder) public
// Claim all the COMP accrued by holder in specific
markets
```

```
function claimComp(address holder, CToken[] memory
cTokens) public
// Claim all the COMP accrued by specific holders in
specific markets for their supplies and/or borrows
function claimComp(address[] memory holders, CToken[]
memory cTokens, bool borrowers, bool suppliers) public
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

troll.claimComp(0x1234...);
```

Web3 1.2.6

```
const comptroller = new
web3.eth.Contract(comptrollerAbi, comptrollerAddress);

await
comptroller.methods.claimComp("0x1234...").send({
from: sender });
```

Market Metadata

The Comptroller contract has an array called `getAllMarkets` that contains the addresses of each cToken contract. Each address in the `getAllMarkets` array can be used to fetch a metadata struct in the Comptroller's markets constant. See the Comptroller Storage contract for the Market struct definition.

Comptroller

```
CToken[] public getAllMarkets;
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

CToken cTokens[] = troll.getAllMarkets();
```

Web3 1.2.6

```
const comptroller = new
web3.eth.Contract(comptrollerAbi, comptrollerAddress);

const cTokens = await
comptroller.methods.getAllMarkets().call();
const cToken = cTokens[0]; // address of a cToken
```

Contract Address

cTokens

Introduction

Each asset supported by the Compound Protocol is integrated through a cToken contract, which is an EIP-20 compliant representation of balances supplied to the protocol. By minting cTokens, users (1) earn interest through the cToken's exchange rate, which increases in value relative to the underlying asset, and (2) gain the ability to use cTokens as collateral.

cTokens are the primary means of interacting with the Compound Protocol; when a user mints, redeems, borrows, repays a borrow, liquidates a borrow, or transfers cTokens, she will do so using the cToken contract.

There are currently two types of cTokens: CErc20 and CEther. Though both types expose the EIP-20 interface, CErc20 wraps an underlying ERC-20 asset, while CEther simply wraps Ether itself. As such, the core functions which involve transferring an asset into the protocol have slightly different interfaces depending on the type, each of which is shown below.

How Do cTokens earn interest?

Do I need to calculate the cToken exchange rate?

Can you walk me through an example?

How do I view my cTokens?

Can I transfer cTokens?

Mint

The mint function transfers an asset into the protocol, which begins accumulating interest based on the current Supply Rate for the asset. The user receives a quantity of cTokens equal to the underlying tokens supplied, divided by the current Exchange Rate.

CErc20

```
function mint(uint mintAmount) returns (uint)
```

- `msg.sender`: The account which shall supply the asset, and own the minted cTokens.
- `mintAmount`: The amount of the asset to be supplied, in units of the underlying asset.
- `RETURN`: 0 on success, otherwise an Error code Before supplying an asset, users must first approve the cToken to access their token balance.

CEther

```
function mint() payable
```

- `msg.value`: The amount of ether to be supplied, in wei.
- `msg.sender`: The account which shall supply the ether, and own the minted cTokens.
- `RETURN`: No return, reverts on error.

Solidity

```
Erc20 underlying = Erc20(0xToken...);     // get a
handle for the underlying asset contract

CErc20 cToken = CErc20(0x3FDA...);        // get a
handle for the corresponding cToken contract
```

```
underlying.approve(address(cToken), 100); // approve
the transfer
assert(cToken.mint(100) == 0);            // mint the
cTokens and assert there is no error
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

await cToken.methods.mint().send({from: myAccount,
value: 50});
```

Redeem

The redeem function converts a specified quantity of cTokens into the underlying asset, and returns them to the user. The amount of underlying tokens received is equal to the quantity of cTokens redeemed, multiplied by the current Exchange Rate. The amount redeemed must be less than the user's Account Liquidity and the market's available liquidity.

CErc20 / CEther

```
function redeem(uint redeemTokens) returns (uint)
```

- `msg.sender`: The account to which redeemed funds shall be transferred.
- `redeemTokens`: The number of cTokens to be redeemed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CEther cToken = CEther(0x3FDB...);

require(cToken.redeem(7) == 0, "something went
wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

cToken.methods.redeem(1).send({from: ...});
```

Redeem Underlying

The redeem underlying function converts cTokens into a specified quantity of the underlying asset, and returns them to the user. The amount of cTokens redeemed is equal to the quantity of underlying tokens received, divided by the current Exchange Rate. The amount redeemed must be less than the user's Account Liquidity and the market's available liquidity.

CErc20 / CEther

```
function redeemUnderlying(uint redeemAmount) returns
(uint)
```

- `msg.sender`: The account to which redeemed funds shall be transferred.
- `redeemAmount`: The amount of underlying to be redeemed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CEther cToken = CEther(0x3FDB...);

require(cToken.redeemUnderlying(50) == 0, "something
went wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

cToken.methods.redeemUnderlying(10).send({from: ...});
```

Borrow

The borrow function transfers an asset from the protocol to the user, and creates a borrow balance which begins accumulating interest based on the Borrow Rate for the asset. The amount borrowed must be less than the user's Account Liquidity and the market's available liquidity. To borrow Ether, the borrower must be 'payable' (solidity).

CErc20 / CEther

```
function borrow(uint borrowAmount) returns (uint)
```

- `msg.sender`: The account to which borrowed funds shall be transferred.
- `borrowAmount` : The amount of the underlying asset to be borrowed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CErc20 cToken = CErc20(0x3FDA...);

require(cToken.borrow(100) == 0, "got collateral?");
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

await cToken.methods.borrow(50).send({from:
0xMyAccount});
```

Repay Borrow

The repay function transfers an asset into the protocol, reducing the user’s borrow balance.

CErc20

```
function repayBorrow(uint repayAmount) returns (uint)
```

- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- `repayAmount`: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. `2^256` - 1) can be used to repay the full amount.
- `RETURN`: 0 on success, otherwise an Error code Before repaying an asset, users must first approve the cToken to access their token balance.

CEther

```
function repayBorrow() payable
```

- `msg.value`: The amount of ether to be repaid, in wei.
- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);

require(cToken.repayBorrow.value(100)() == 0,
"transfer approved?");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

cToken.methods.repayBorrow(10000).send({from: ...});
```

Repay Borrow Behalf

The repay function transfers an asset into the protocol, reducing the target user’s borrow balance.

CErc20

```
function repayBorrowBehalf(address borrower, uint
repayAmount) returns (uint)
```

- `msg.sender`: The account which shall repay the borrow.

- `borrower`: The account which borrowed the asset to be repaid.
- `repayAmount`: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. `2^256` - 1) can be used to repay the full amount.
- `RETURN`: 0 on success, otherwise an Error code Before repaying an asset, users must first approve the cToken to access their token balance.

CEther

```
function repayBorrowBehalf(address borrower) payable
```

- `msg.value`: The amount of ether to be repaid, in wei.
- `msg.sender`: The account which shall repay the borrow.
- `borrower`: The account which borrowed the asset to be repaid.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);

require(cToken.repayBorrowBehalf.value(100)(0xBorrower
) == 0, "transfer approved?");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

await cToken.methods.repayBorrowBehalf(0xBorrower,
10000).send({from: 0xPayer});
```

Transfer

Transfer is an ERC-20 method that allows accounts to send tokens to other Ethereum addresses. A cToken transfer will fail if the account has entered that cToken market and the transfer would have put the account into a state of negative liquidity.

CErc20 / CEther

```
function transfer(address recipient, uint256 amount)
returns (bool)
```

- `recipient`: The transfer recipient address.
- `amount`: The amount of cTokens to transfer.
- `RETURN`: Returns a boolean value indicating whether or not the operation succeeded.

Solidity

```
CEther cToken = CEther(0x3FDB...);

cToken.transfer(0xABCD..., 100000000000);
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

await cToken.methods.transfer(0xABCD...,
100000000000).send({from: 0xSender});
```

Liquidate Borrow

A user who has negative account liquidity is subject to liquidation by other users of the protocol to return his/her account liquidity back to positive (i.e. above the collateral requirement). When a liquidation occurs, a liquidator may repay some or all of an outstanding borrow on behalf of a borrower and in return receive a discounted amount of collateral held by the borrower; this discount is defined as the liquidation incentive. A liquidator may close up to a certain fixed percentage (i.e. close factor) of any individual outstanding borrow of the underwater account. Unlike in v1, liquidators must interact with each cToken contract in which they wish to repay a borrow and seize another asset as collateral. When collateral is seized, the liquidator is transferred cTokens, which they may redeem the same as if they had supplied the asset themselves. Users must approve each cToken contract before calling liquidate (i.e. on the borrowed asset which they are repaying), as they are transferring funds into the contract.

CErc20

```
function liquidateBorrow(address borrower, uint
amount, address collateral) returns (uint)
```

- `msg.sender`: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- `borrower`: The account with negative account liquidity that shall be liquidated.
- `repayAmount`: The amount of the borrowed asset to be repaid and converted into collateral, specified in units of the underlying borrowed asset.
- `cTokenCollateral`: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- `RETURN`: 0 on success, otherwise an Error code Before supplying an asset, users must first approve the cToken to access their token balance.

CEther

```
function liquidateBorrow(address borrower, address
cTokenCollateral) payable
```

- `msg.value`: The amount of ether to be repaid and converted into collateral, in wei.
- `msg.sender`: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- `borrower`: The account with negative account liquidity that shall be liquidated.
- `cTokenCollateral`: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);

CErc20 cTokenCollateral = CErc20(0x3FDA...);
require(cToken.liquidateBorrow.value(100)(0xBorrower,
cTokenCollateral) == 0, "borrower underwater??");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);

const cTokenCollateral = CEther.at(0x3FDB...);
await cToken.methods.liquidateBorrow(0xBorrower, 33,
cTokenCollateral).send({from: 0xLiquidator});
```

Key Events

Event	Description
`Mint(address minter, uint mintAmount, uint mintTokens)`	Emitted upon a successful Mint.
`Redeem(address redeemer, uint redeemAmount, uint redeemTokens)`	Emitted upon a successful Redeem.
`Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows)`	Emitted upon a successful Borrow.
`RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint totalBorrows)`	Emitted upon a successful Repay Borrow.
`LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address cTokenCollateral, uint seizeTokens)`	Emitted upon a successful Liquidate Borrow.

Error Codes

Code	Name	Description
0	`NO_ERROR`	Not a failure.

Code	Name	Description
1	`UNAUTHORIZED`	The sender is not authorized to perform this action.
2	`BAD_INPUT`	An invalid argument was supplied by the caller.
3	`COMPTROLLER_REJECTION`	The action would violate the comptroller policy.
4	`COMPTROLLER_CALCULATION_ERROR`	An internal calculation has failed in the comptroller.
5	`INTEREST_RATE_MODEL_ERROR`	The interest rate model returned an invalid value.
6	`INVALID_ACCOUNT_PAIR`	The specified combination of accounts is invalid.
7	`INVALID_CLOSE_AMOUNT_REQUESTED`	The amount to liquidate is invalid.
8	`INVALID_COLLATERAL_FACTOR`	The collateral factor is invalid.
9	`MATH_ERROR`	A math calculation error occurred.
10	`MARKET_NOT_FRESH`	Interest has not been properly accrued.

Code	Name	Description
11	`MARKET_NOT_LISTED`	The market is not currently listed by its comptroller.
12	`TOKEN_INSUFFICIENT_ALLOWANCE`	ERC-20 contract must *allow* Money Market contract to call `transferFrom`. The current allowance is either 0 or less than the requested supply, repayBorrow or liquidate amount.
13	`TOKEN_INSUFFICIENT_BALANCE`	Caller does not have sufficient balance in the ERC-20 contract to complete the desired action.
14	`TOKEN_INSUFFICIENT_CASH`	The market does not have a sufficient cash balance to complete the transaction. You may attempt this transaction again later.
15	`TOKEN_TRANSFER_IN_FAILED`	Failure in ERC-20 when transfering token into the market.
16	`TOKEN_TRANSFER_OUT_FAILED`	Failure in ERC-20 when transfering token out of the market.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED
2	ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED
3	ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED
4	ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED
5	ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED
6	ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED
7	BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
8	BORROW_ACCRUE_INTEREST_FAILED
9	BORROW_CASH_NOT_AVAILABLE
10	BORROW_FRESHNESS_CHECK
11	BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED
12	BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED

Code	Name
13	BORROW_MARKET_NOT_LISTED
14	BORROW_COMPTROLLER_REJECTION
15	LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED
16	LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED
17	LIQUIDATE_COLLATERAL_FRESHNESS_CHECK
18	LIQUIDATE_COMPTROLLER_REJECTION
19	LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED
20	LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX
21	LIQUIDATE_CLOSE_AMOUNT_IS_ZERO
22	LIQUIDATE_FRESHNESS_CHECK
23	LIQUIDATE_LIQUIDATOR_IS_BORROWER
24	LIQUIDATE_REPAY_BORROW_FRESH_FAILED
25	LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED
26	LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED

Code	Name
27	LIQUIDATE_SEIZE_COMPTROLLER_REJECTION
28	LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER
29	LIQUIDATE_SEIZE_TOO_MUCH
30	MINT_ACCRUE_INTEREST_FAILED
31	MINT_COMPTROLLER_REJECTION
32	MINT_EXCHANGE_CALCULATION_FAILED
33	MINT_EXCHANGE_RATE_READ_FAILED
34	MINT_FRESHNESS_CHECK
35	MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED
36	MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED
37	MINT_TRANSFER_IN_FAILED
38	MINT_TRANSFER_IN_NOT_POSSIBLE
39	REDEEM_ACCRUE_INTEREST_FAILED
40	REDEEM_COMPTROLLER_REJECTION

Code	Name
41	REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED
42	REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED
43	REDEEM_EXCHANGE_RATE_READ_FAILED
44	REDEEM_FRESHNESS_CHECK
45	REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED
46	REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED
47	REDEEM_TRANSFER_OUT_NOT_POSSIBLE
48	REDUCE_RESERVES_ACCRUE_INTEREST_FAILED
49	REDUCE_RESERVES_ADMIN_CHECK
50	REDUCE_RESERVES_CASH_NOT_AVAILABLE
51	REDUCE_RESERVES_FRESH_CHECK
52	REDUCE_RESERVES_VALIDATION
53	REPAY_BEHALF_ACCRUE_INTEREST_FAILED
54	REPAY_BORROW_ACCRUE_INTEREST_FAILED

Code	Name
55	REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
56	REPAY_BORROW_COMPTROLLER_REJECTION
57	REPAY_BORROW_FRESHNESS_CHECK
58	REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED
59	REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED
60	REPAY_BORROW_TRANSFER_IN_NOT_POSSIBLE
61	SET_COLLATERAL_FACTOR_OWNER_CHECK
62	SET_COLLATERAL_FACTOR_VALIDATION
63	SET_COMPTROLLER_OWNER_CHECK
64	SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED
65	SET_INTEREST_RATE_MODEL_FRESH_CHECK
66	SET_INTEREST_RATE_MODEL_OWNER_CHECK
67	SET_MAX_ASSETS_OWNER_CHECK
68	SET_ORACLE_MARKET_NOT_LISTED

Code	Name
69	SET_PENDING_ADMIN_OWNER_CHECK
70	SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED
71	SET_RESERVE_FACTOR_ADMIN_CHECK
72	SET_RESERVE_FACTOR_FRESH_CHECK
73	SET_RESERVE_FACTOR_BOUNDS_CHECK
74	TRANSFER_COMPTROLLER_REJECTION
75	TRANSFER_NOT_ALLOWED
76	TRANSFER_NOT_ENOUGH
77	TRANSFER_TOO_MUCH

Exchange Rate

Each cToken is convertible into an ever increasing quantity of the underlying asset, as interest accrues in the market. The exchange rate between a cToken and the underlying asset is equal to:

```
exchangeRate = (getCash() + totalBorrows() -
totalReserves()) / totalSupply()
```

CErc20 / CEther

```
function exchangeRateCurrent() returns (uint)
```

- `RETURN`: The current exchange rate as an unsigned integer, scaled by 1 * 10^(18 - 8 + Underlying Token Decimals).

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint exchangeRateMantissa =
cToken.exchangeRateCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const exchangeRate = (await
cToken.methods.exchangeRateCurrent().call()) / 1e18;
```

Tip: note the use of `call` vs. `send` to invoke the function from off-chain without incurring gas costs.

Get Cash

Cash is the amount of underlying balance owned by this cToken contract. One may query the total amount of cash currently available to this market.

CErc20 / CEther

```
function getCash() returns (uint)
```

- `RETURN`: The quantity of underlying asset owned by the contract.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint cash = cToken.getCash();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const cash = (await cToken.methods.getCash().call());
```

Total Borrows

Total Borrows is the amount of underlying currently loaned out by the market, and the amount upon which interest is accumulated to suppliers of the market.

CErc20 / CEther

```
function totalBorrowsCurrent() returns (uint)
```

- `RETURN`: The total amount of borrowed underlying, with interest.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint borrows = cToken.totalBorrowsCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const borrows = (await
cToken.methods.totalBorrowsCurrent().call());
```

Borrow Balance

A user who borrows assets from the protocol is subject to accumulated interest based on the current borrow rate. Interest is accumulated every block and integrations may use this function to obtain the current value of a user’s borrow balance with interest.

CErc20 / CEther

```
function borrowBalanceCurrent(address account) returns
(uint)
```

- `account`: The account which borrowed the assets.
- `RETURN`: The user’s current borrow balance (with interest) in units of the underlying asset.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint borrows =
cToken.borrowBalanceCurrent(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const borrows = await
cToken.methods.borrowBalanceCurrent(account).call();
```

Borrow Rate

At any point in time one may query the contract to get the current borrow rate per block.

CErc20 / CEther

```
function borrowRatePerBlock() returns (uint)
```

- `RETURN`: The current borrow rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint borrowRateMantissa = cToken.borrowRatePerBlock();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const borrowRate = (await
cToken.methods.borrowRatePerBlock().call()) / 1e18;
```

Total Supply

Total Supply is the number of tokens currently in circulation in this cToken market. It is part of the EIP-20 interface of the cToken contract.

CErc20 / CEther

```
function totalSupply() returns (uint)
```

- `RETURN`: The total number of tokens in circulation for the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint tokens = cToken.totalSupply();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const tokens = (await
cToken.methods.totalSupply().call());
```

Underlying Balance

The user’s underlying balance, representing their assets in the protocol, is equal to the user’s cToken balance multiplied by the Exchange Rate.

CErc20 / CEther

```
function balanceOfUnderlying(address account) returns
(uint)
```

- `account`: The account to get the underlying balance of.
- `RETURN`: The amount of underlying currently owned by the account.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint tokens = cToken.balanceOfUnderlying(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const tokens = await
cToken.methods.balanceOfUnderlying(account).call();
```

Supply Rate

At any point in time one may query the contract to get the current supply rate per block. The supply rate is derived from the borrow rate, reserve factor and the amount of total borrows.

CErc20 / CEther

```
function supplyRatePerBlock() returns (uint)
```

- `RETURN`: The current supply rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint supplyRateMantissa = cToken.supplyRatePerBlock();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const supplyRate = (await
cToken.methods.supplyRatePerBlock().call()) / 1e18;
```

Total Reserves

Reserves are an accounting entry in each cToken contract that represents a portion of historical interest set aside as cash which can be withdrawn or transferred through the protocol’s governance. A small portion of borrower interest accrues into the protocol, determined by the reserve factor.

CErc20 / CEther

```
function totalReserves() returns (uint)
```

- `RETURN`: The total amount of reserves held in the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint reserves = cToken.totalReserves();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const reserves = (await
cToken.methods.totalReserves().call());
```

Reserve Factor

The reserve factor defines the portion of borrower interest that is converted into reserves.

CErc20 / CEther

```
function reserveFactorMantissa() returns (uint)
```

- `RETURN`: The current reserve factor as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);

uint reserveFactorMantissa =
cToken.reserveFactorMantissa();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);

const reserveFactor = (await
cToken.methods.reserveFactorMantissa().call()) / 1e18;
```

Contract Address

Please enable cookies.

Email Protection

You are unable to access this email address compound.finance

The website from which you got to this page is protected by Cloudflare. Email addresses on that page have been hidden in order to keep them from being accessed by malicious bots. **You must enable Javascript in your browser in order to decode the e-mail address**.

If you have a website and are interested in protecting it in a similar way, you can sign up for Cloudflare.

- How does Cloudflare protect email addresses on website from spammers?
- Can I sign up for Cloudflare?

Cloudflare Ray ID: **7dc981b1bd4136cf** • Your IP: Click to reveal 216.249.57.2 • Performance & security by Cloudflare

Governance

Introduction

The Compound protocol is governed and upgraded by COMP token-holders, using three distinct components; the COMP token, governance module (Governor Bravo), and Timelock. Together, these contracts allow the community to propose, vote, and implement changes through the administrative functions of a cToken or the Comptroller. Proposals can modify system parameters, support new markets, or add entirely new functionality to the protocol.

COMP token-holders can delegate their voting rights to themselves, or an address of their choice. Addresses delegated at least 25,000 COMP can create governance proposals; any address can lock 100 COMP to create an Autonomous Proposal, which becomes a governance proposal after being delegated 25,000 COMP.

When a governance proposal is created, it enters a 2 day review period, after which voting weights are recorded and voting begins. Voting lasts for 3 days; if a majority, and at least 400,000 votes are cast for the proposal, it is queued in the Timelock, and can be implemented 2 days later. In total, any change to the protocol takes at least one week.

COMP

COMP is an ERC-20 token that allows the owner to delegate voting rights to any address, including their own address. Changes to the owner’s token balance automatically adjust the voting rights of the delegate.

Delegate

Delegate votes from the sender to the delegatee. Users can delegate to 1 address at a time, and the number of votes added to the delegatee’s vote count is equivalent to the balance of COMP in the user’s account. Votes are delegated from the current block and onward, until the sender delegates again, or transfers their COMP.

COMP

```
function delegate(address delegatee)
```

- `delegatee`: The address in which the sender wishes to delegate their votes to.
- `msg.sender`: The address of the COMP token holder that is attempting to delegate their votes.
- `RETURN`: No return, reverts on error.

Solidity

```
Comp comp = Comp(0x123...); // contract address

comp.delegate(delegateeAddress);
```

Web3 1.2.6

```
const tx = await
comp.methods.delegate(delegateeAddress).send({ from:
sender });
```

Delegate By Signature

Delegate votes from the signatory to the delegatee. This method has the same purpose as Delegate but it instead enables offline signatures to participate in Compound governance vote delegation. For more details on how to create an offline signature, review EIP-712.

COMP

```
function delegateBySig(address delegatee, uint nonce,
uint expiry, uint8 v, bytes32 r, bytes32 s)
```

- `delegatee`: The address in which the sender wishes to delegate their votes to.
- `nonce`: The contract state required to match the signature. This can be retrieved from the contract’s public nonces mapping.
- `expiry`: The time at which to expire the signature. A block timestamp as seconds since the unix epoch (uint).
- `v`: The recovery byte of the signature.
- `r`: Half of the ECDSA signature pair.
- `s`: Half of the ECDSA signature pair.
- `RETURN`: No return, reverts on error.

Solidity

```
Comp comp = Comp(0x123...); // contract address
```

```
comp.delegateBySig(delegateeAddress, nonce, expiry, v,
r, s);
```

Web3 1.2.6

```
const tx = await
comp.methods.delegateBySig(delegateeAddress, nonce,
expiry, v, r, s).send({});
```

Get Current Votes

Gets the balance of votes for an account as of the current block.

COMP

```
function getCurrentVotes(address account) returns
(uint96)
```

- `account`: Address of the account in which to retrieve the number of votes.
- `RETURN`: The number of votes (integer).

Solidity

```
Comp comp = Comp(0x123...); // contract address

uint votes = comp.getCurrentVotes(0xabc...);
```

Web3 1.2.6

```
const account = '0x123...'; // contract address
const votes = await
comp.methods.getCurrentVotes(account).call();
```

Get Prior Votes

Gets the prior number of votes for an account at a specific block number. The block number passed must be a finalized block or the function will revert.

COMP

```
function getPriorVotes(address account, uint
blockNumber) returns (uint96)
```

- `account`: Address of the account in which to retrieve the prior number of votes.
- `blockNumber`: The block number at which to retrieve the prior number of votes.
- `RETURN`: The number of prior votes.

Solidity

```
Comp comp = Comp(0x123...); // contract address

uint priorVotes = comp.getPriorVotes(account,
blockNumber);
```

Web3 1.2.6

```
const priorVotes = await
comp.methods.getPriorVotes(account,
blockNumber).call();
```

Key Events

Event	Description
`DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate)`	An event thats emitted when an account changes its delegate.
`DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance)`	An event thats emitted when a delegate account's vote balance changes.
`ProposalCreated(uint id, address proposer, address[] targets, uint[] values, string[] signatures, bytes[] calldatas, uint startBlock, uint endBlock, string description)`	An event emitted when a new proposal is created.
`VoteCast(address voter, uint proposalId, bool support, uint votes)`	An event emitted when a vote has been cast on a proposal.
`ProposalCanceled(uint id)`	An event emitted when a proposal

Event	Description
	has been canceled.
`ProposalQueued(uint id, uint eta)`	An event emitted when a proposal has been queued in the Timelock.
`ProposalExecuted(uint id)`	An event emitted when a proposal has been executed in the Timelock.

Governor Bravo

Governor Bravo is the governance module of the protocol; it allows addresses with more than 25,000 COMP to propose changes to the protocol. Addresses that held voting weight, at the start of the proposal, invoked through the getpriorvotes function, can submit their votes during a 3 day voting period. If a majority, and at least 400,000 votes are cast for the proposal, it is queued in the Timelock, and can be implemented after 2 days.

Quorum Votes

The required minimum number of votes in support of a proposal for it to succeed.

Governor Bravo

```
function quorumVotes() public pure returns (uint)
```

- `RETURN`: The minimum number of votes required for a proposal to succeed.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint quorum = gov.quorumVotes();
```

Web3 1.2.6

```
const quorum = await gov.methods.quorumVotes().call();
```

Proposal Threshold

The minimum number of votes required for an account to create a proposal. This can be changed through governance.

Governor Bravo

```
function proposalThreshold() returns (uint)
```

- `RETURN`: The minimum number of votes required for an account to create a proposal.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint threshold = gov.proposalThreshold();
```

Web3 1.2.6

```
const threshold = await
gov.methods.proposalThreshold().call();
```

Proposal Max Operations

The maximum number of actions that can be included in a proposal. Actions are functions calls that will be made when a proposal succeeds and executes.

Governor Bravo

```
function proposalMaxOperations() returns (uint)
```

- `RETURN`: The maximum number of actions that can be included in a proposal.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint operations = gov.proposalMaxOperations();
```

Web3 1.2.6

```
const operations = await
gov.methods.proposalMaxOperations().call();
```

Voting Delay

The number of Ethereum blocks to wait before voting on a proposal may begin. This value is added to the current block number when a proposal is created. This can be changed through governance.

Governor Bravo

```
function votingDelay() returns (uint)
```

- `RETURN`: Number of blocks to wait before voting on a proposal may begin.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint blocks = gov.votingDelay();
```

Web3 1.2.6

```
const blocks = await gov.methods.votingDelay().call();
```

Voting Period

The duration of voting on a proposal, in Ethereum blocks. This can be changed through governance.

Governor Bravo

```
function votingPeriod() returns (uint)
```

- `RETURN`: The duration of voting on a proposal, in Ethereum blocks.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint blocks = gov.votingPeriod();
```

Web3 1.2.6

```
const blocks = await
gov.methods.votingPeriod().call();
```

Propose

Create a Proposal to change the protocol. E.g., A proposal can set a cToken's interest rate model or risk parameters on the Comptroller. Proposals will be voted on by delegated voters. If there is sufficient support before the voting period ends, the proposal shall be automatically enacted. Enacted proposals are queued and executed in the Compound Timelock contract.

The sender must hold more COMP than the current proposal threshold (`proposalThreshold()`) as of the immediately previous block. If the threshold is 25,000 COMP, the sender must have been delegated more than 1% of all COMP in order to create a proposal. The proposal can have up to 10 actions (based on `proposalMaxOperations()`).

The proposer cannot create another proposal if they currently have a pending or active proposal. It is not possible to queue two identical actions in the same block (due to a restriction in the Timelock), therefore actions in a single proposal must be unique, and unique proposals that share an identical action must be queued in different blocks.

Governor Bravo

```
function propose(address[] memory targets, uint[]
memory values, string[] memory signatures, bytes[]
memory calldatas, string memory description) returns
(uint)
```

- `targets`: The ordered list of target addresses for calls to be made during proposal execution. This array must be the same length as all other array parameters in this function.
- `values`: The ordered list of values (i.e. msg.value) to be passed to the calls made during proposal execution. This array must be the same length as all other array parameters in this function.

- `signatures`: The ordered list of function signatures to be passed during execution. This array must be the same length as all other array parameters in this function.
- `calldatas`: The ordered list of data to be passed to each individual function call during proposal execution. This array must be the same length as all other array parameters in this function.
- `description`: A human readable description of the proposal and the changes it will enact.
- `RETURN`: The ID of the newly created proposal.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint proposalId = gov.propose(targets, values,
signatures, calldatas, description);
```

Web3 1.2.6

```
const tx = gov.methods.propose(targets, values,
signatures, calldatas, description).send({ from:
sender });
```

Queue

After a proposal has succeeded, it is moved into the Timelock waiting period using this function. The waiting period (e.g. 2 days) begins when this function is called. The queue function can be called by any Ethereum address.

Governor Bravo

```
function queue(uint proposalId)
```

- `proposalId`: ID of a proposal that has succeeded.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.queue(proposalId);
```

Web3 1.2.6

```
const tx = gov.methods.queue(proposalId).send({ from:
sender });
```

Execute

After the Timelock waiting period has elapsed, a proposal can be executed using this function, which applies the proposal changes to the target contracts. This will invoke each of the actions described in the proposal. The execute function can be called by any Ethereum address. Note: this function is *payable*, so the Timelock contract can invoke payable functions that were selected in the proposal.

Governor Bravo

```
function execute(uint proposalId) payable
```

- `proposalId`: ID of a succeeded proposal to execute.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.execute(proposalId).value(999).gas(999)();
```

Web3 1.2.6

```
const tx = gov.methods.execute(proposalId).send({
from: sender, value: 1 });
```

Cancel

A proposal is eligible to be cancelled at any time prior to its execution, including while queued in the Timelock, using this function.

The cancel function can be called by the proposal creator, or any Ethereum address, if the proposal creator fails to maintain more delegated votes than the proposal threshold (e.g. 25,000).

Governor Bravo

```
function cancel(uint proposalId)
```

- `proposalId`: ID of a proposal to cancel. The proposal cannot have already been executed.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.cancel(proposalId);
```

Web3 1.2.6

```
const tx = gov.methods.cancel(proposalId).send({ from:
sender });
```

Get Actions

Gets the actions of a selected proposal. Pass a proposal ID and get the targets, values, signatures and calldatas of that proposal.

Governor Bravo

```
function getActions(uint proposalId) returns (uint
proposalId) public view returns (address[] memory
targets, uint[] memory values, string[] memory
signatures, bytes[] memory calldatas)
```

- `proposalId`: ID of a proposal in which to get its actions.
- `RETURN`: Reverts if the proposal ID is invalid. If successful, the following 4 references are returned.
    - Array of addresses of contracts the proposal calls.
    - Array of unsigned integers the proposal uses as values.
    - Array of strings of the proposal’s signatures.
    - Array of calldata bytes of the proposal.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

uint proposalId = 123;
(address[] memory targets, uint[] memory values,
string[] memory signatures, bytes[] memory calldatas)
= gov.getActions(proposalId);
```

Web3 1.2.6

```
const {0: targets, 1: values, 2: signatures, 3:
calldatas} =
gov.methods.getActions(proposalId).call();
```

Get Receipt

Gets a proposal ballot receipt of the indicated voter.

Governor Bravo

```
function getReceipt(uint proposalId, address voter)
returns (Receipt memory)
```

- `proposalId`: ID of the proposal in which to get a voter’s ballot receipt.
- `voter`: Address of the account of a proposal voter.
- `RETURN`: Reverts on error. If successful, returns a Receipt struct for the ballot of the voter address.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

Receipt ballot = gov.getReceipt(proposalId,
voterAddress);
```

Web3 1.2.6

```
const proposalId = 11;

const voterAddress = '0x123...';
const result = await
gov.methods.getReceipt(proposalId,
voterAddress).call();
const { hasVoted, support, votes } = result;
```

State

Gets the proposal state for the specified proposal. The return value, `ProposalState` is an enumerated type defined in the Governor Bravo contract.

Governor Bravo

```
function state(uint proposalId) returns
(ProposalState)
```

- `proposalId`: ID of a proposal in which to get its state.
- `RETURN`: Enumerated type ProposalState. The types are Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired, andExecuted.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

GovernorBravo.ProposalState state = gov.state(123);
```

Web3 1.2.6

```
const proposalStates = ['Pending', 'Active',
'Canceled', 'Defeated', 'Succeeded', 'Queued',
'Expired', 'Executed'];

const proposalId = 123;
result = await gov.methods.state(proposalId).call();
const proposalState = proposalStates[result];
```

Cast Vote

Cast a vote on a proposal. The account’s voting weight is determined by the number of votes the account had delegated to it at the time the proposal state became active.

Governor Bravo

```
function castVote(uint proposalId, uint8 support)
```

- `proposalId`: ID of a proposal in which to cast a vote.
- `support`: An integer of 0 for against, 1 for in-favor, and 2 for abstain.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.castVote(proposalId, 1);
```

Web3 1.2.6

```
const tx = gov.methods.castVote(proposalId, 0).send({
from: sender });
```

Cast Vote With Reason

Cast a vote on a proposal with a reason attached to the vote.

Governor Bravo

```
function castVoteWithReason(uint proposalId, uint8
support, string calldata reason)
```

- `proposalId`: ID of a proposal in which to cast a vote.
- `support`: An integer of 0 for against, 1 for in-favor, and 2 for abstain.
- `reason`: A string containing the voter’s reason for their vote selection.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.castVoteWithReason(proposalId, 2, "I think...");
```

Web3 1.2.6

```
const tx = gov.methods.castVoteWithReason(proposalId,
0, "I think...").send({ from: sender });
```

Cast Vote By Signature

Cast a vote on a proposal. The account’s voting weight is determined by the number of votes the account had delegated at the time that proposal state became active. This method has the same purpose as Cast Vote but it instead enables offline signatures to participate in Compound governance voting. For more details on how to create an offline signature, review EIP-712.

Governor Bravo

```
function castVoteBySig(uint proposalId, uint8 support,
uint8 v, bytes32 r, bytes32 s)
```

- `proposalId`: ID of a proposal in which to cast a vote.
- `support`: An integer of 0 for against, 1 for in-favor, and 2 for abstain.
- `v`: The recovery byte of the signature.
- `r`: Half of the ECDSA signature pair.
- `s`: Half of the ECDSA signature pair.
- `RETURN`: No return, reverts on error.

Solidity

```
GovernorBravo gov = GovernorBravo(0x123...); //
contract address

gov.castVoteBySig(proposalId, 0, v, r, s);
```

Web3 1.2.6

```
const tx = await gov.methods.castVoteBySig(proposalId,
1, v, r, s).send({});
```

Timelock

Each protocol contract is controlled by the Timelock contract, which can modify system parameters, logic, and contracts in a ‘time-delayed, opt-out’ upgrade pattern. The Timelock has a hard-coded minimum delay which is the least amount of notice possible for a governance action. The Timelock contract queues and executes proposals that have passed a Governance vote.

Pause Guardian

The Comptroller contract designates a Pause Guardian address capable of disabling protocol functionality. Used only in the event of an unforeseen vulnerability, the Pause Guardian has one and only one ability: to disable a select set of functions: Mint, Borrow, Transfer, and Liquidate. The Pause Guardian cannot unpause an action, nor can it ever prevent users from calling Redeem, or Repay Borrow to close positions and exit the protocol. COMP token-holders designate the Pause Guardian address, which is held by the Community Multi-Sig.

Contract	Address

- Compound III
- Interest Rates
- Collateral & Borrowing
- Liquidation
- Account Management
- Protocol Rewards
- Governance
- Helper Functions

Compound III

Interest Rates
Collateral & Borrowing
Liquidation
Account Management
Protocol Rewards
Governance
Helper Functions

Governance

Compound III is a decentralized protocol that is governed by holders and delegates of COMP. Governance allows the community to propose, vote, and implement changes through the administrative smart contract functions of the Compound III protocol. For more information on the Governor and Timelock see the original governance section.

All instances of Compound III are controlled by the Timelock contract which is the same administrator of the Compound v2 protocol. The governance system has control over each *proxy*, the *Configurator implementation*, the *Comet factory*, and the *Comet implementation*.

Each time an immutable parameter is set via governance proposal, a new Comet implementation must be deployed by the Comet factory. If the proposal is approved by the community, the proxy will point to the new implementation upon execution.

To set specific protocol parameters in a proposal, the Timelock must call all of the relevant set methods on the *Configurator* contract, followed by `deployAndUpgradeTo` on the *CometProxyAdmin* contract.

Multi-chain Governance

The Compound III protocol can be deployed on any EVM chain. The deployment must have access to on-chain asset prices and governance messages passed from Ethereum Mainnet. The Timelock on Mainnet is the administrator of all community sanctioned instances of Compound III.

Each deployment outside of Mainnet needs to have a Bridge Receiver and Local Timelock contract on its chain. Governance proposals executed on Mainnet must be read by the chain's bridge and published to the Bridge Receiver. Local Timelocks have an additional delay before Comet admin functions can be called via proposal execution.

Compound III instance initializations are logged on-chain using the ENS text record system. The text record can only be modified by a Governance proposal. It can be viewed at v3-additional-grants.compound-community-licenses.eth when the browser network is set to Ethereum Mainnet.

Set Comet Factory

This function sets the official contract address of the Comet factory. The only acceptable caller is the Governor.

Configurator

```
function setFactory(address cometProxy, address
newFactory) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newFactory`: The address of the new Comet contract factory.
- `RETURN`: No return, reverts on error.

Set Governor

This function sets the official contract address of the Compound III protocol Governor for subsequent proposals.

Configurator

```
function setGovernor(address cometProxy, address
newGovernor) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newGovernor`: The address of the new Compound III Governor.
- `RETURN`: No return, reverts on error.

Set Pause Guardian

This function sets the official contract address of the Compound III protocol pause guardian. This address has the power to pause supply, transfer, withdraw, absorb, and buy collateral operations within Compound III.

COMP token-holders designate the Pause Guardian address, which is held by the Community Multi-Sig.

Configurator

```
function setPauseGuardian(address cometProxy, address
newPauseGuardian) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newPauseGuardian`: The address of the new pause guardian.
- `RETURN`: No return, reverts on error.

Pause Protocol Functionality

This function pauses the specified protocol functionality in the event of an unforeseen vulnerability. The only addresses that are allowed to call this function are the Governor and the Pause Guardian.

Comet

```
function pause(

    bool supplyPaused,
    bool transferPaused,
    bool withdrawPaused,
    bool absorbPaused,
    bool buyPaused
) override external
```

- `supplyPaused`: Enables or disables all accounts' ability to supply assets to the protocol.
- `transferPaused`: Enables or disables all account's ability to transfer assets within the protocol.
- `withdrawPaused`: Enables or disables all account's ability to withdraw assets from the protocol.
- `absorbPaused`: Enables or disables protocol absorptions.

- `buyPaused`: Enables or disables the protocol’s ability to sell absorbed collateral.
- `RETURN`: No return, reverts on error.

Is Supply Paused

This function returns a boolean indicating whether or not the protocol supply functionality is presently paused.

Comet

```
function isSupplyPaused() override public view returns
(bool)
```

- `RETURN`: A boolean value of whether or not the protocol functionality is presently paused.

Is Transfer Paused

This function returns a boolean indicating whether or not the protocol transfer functionality is presently paused.

Comet

```
function isTransferPaused() override public view
returns (bool)
```

- `RETURN`: A boolean value of whether or not the protocol functionality is presently paused.

Is Withdraw Paused

This function returns a boolean indicating whether or not the protocol withdraw functionality is presently paused.

Comet

```
function isWithdrawPaused() override public view
returns (bool)
```

- `RETURN`: A boolean value of whether or not the protocol functionality is presently paused.

Is Absorb Paused

This function returns a boolean indicating whether or not the protocol absorb functionality is presently paused.

Comet

```
function isAbsorbPaused() override public view returns
(bool)
```

- `RETURN`: A boolean value of whether or not the protocol functionality is presently paused.

Is Buy Paused

This function returns a boolean indicating whether or not the protocol’s selling of absorbed collateral functionality is presently paused.

Comet

```
function isBuyPaused() override public view returns
(bool)
```

- `RETURN`: A boolean value of whether or not the protocol functionality is presently paused.

Set Base Token Price Feed

This function sets the official contract address of the price feed of the protocol base asset.

Configurator

```
function setBaseTokenPriceFeed(address cometProxy,
address newBaseTokenPriceFeed) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newBaseTokenPriceFeed`: The address of the new price feed contract.
- `RETURN`: No return, reverts on error.

Set Extension Delegate

This function sets the official contract address of the protocol’s Comet extension delegate. The methods in **CometExt.sol** are able to be called via the same proxy as **Comet.sol**.

Configurator

```
function setExtensionDelegate(address cometProxy,
address newExtensionDelegate) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newExtensionDelegate`: The address of the new extension delegate contract.
- `RETURN`: No return, reverts on error.

Set Borrow Kink

This function sets the borrow interest rate utilization curve kink for the Compound III base asset.

Configurator

```
function setBorrowKink(address cometProxy, uint64
newKink) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newKink`: The new kink parameter.
- `RETURN`: No return, reverts on error.

Set Borrow Interest Rate Slope (Low)

This function sets the borrow interest rate slope low bound in the approximate amount of seconds in one year.

Configurator

```
function setBorrowPerYearInterestRateSlopeLow(address
cometProxy, uint64 newSlope) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope low bound as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Borrow Interest Rate Slope (High)

This function sets the borrow interest rate slope high bound in the approximate amount of seconds in one year.

Configurator

```
function setBorrowPerYearInterestRateSlopeHigh(address
cometProxy, uint64 newSlope) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope high bound as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Borrow Interest Rate Slope (Base)

This function sets the borrow interest rate slope base in the approximate amount of seconds in one year.

Configurator

```
function setBorrowPerYearInterestRateBase(address
cometProxy, uint64 newBase) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope base as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Supply Kink

This function sets the supply interest rate utilization curve kink for the Compound III base asset.

Configurator

```
function setSupplyKink(address cometProxy, uint64
newKink) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newKink`: The new kink parameter.
- `RETURN`: No return, reverts on error.

Set Supply Interest Rate Slope (Low)

This function sets the supply interest rate slope low bound in the approximate amount of seconds in one year.

Configurator

```
function setSupplyPerYearInterestRateSlopeLow(address
cometProxy, uint64 newSlope) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope low bound as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Supply Interest Rate Slope (High)

This function sets the supply interest rate slope high bound in the approximate amount of seconds in one year.

Configurator

```
function setSupplyPerYearInterestRateSlopeHigh(address
cometProxy, uint64 newSlope) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope high bound as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Supply Interest Rate Slope (Base)

This function sets the supply interest rate slope base in the approximate amount of seconds in one year.

Configurator

```
function setSupplyPerYearInterestRateBase(address
cometProxy, uint64 newBase) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newSlope`: The slope base as an unsigned integer.
- `RETURN`: No return, reverts on error.

Set Store Front Price Factor

This function sets the fraction of the liquidation penalty that goes to buyers of collateral instead of the protocol. This factor is used to calculate the discount rate of collateral for sale as part of the account absorption process. The rate is a decimal scaled up by `10 ^ 18`.

Configurator

```
function setStoreFrontPriceFactor(address cometProxy,
uint64 newStoreFrontPriceFactor) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newStoreFrontPriceFactor`: The new price factor as an unsigned integer expressed as a decimal scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Set Base Tracking Supply Speed

This function sets the rate at which base asset supplier accounts accrue rewards.

Configurator

```
function setBaseTrackingSupplySpeed(address
cometProxy, uint64 newBaseTrackingSupplySpeed)
external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newBaseTrackingSupplySpeed`: The rate as an APR expressed as a decimal scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Set Base Tracking Borrow Speed

This function sets the rate at which base asset borrower accounts accrue rewards.

Configurator

```
function setBaseTrackingBorrowSpeed(address
cometProxy, uint64 newBaseTrackingBorrowSpeed)
external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newBaseTrackingBorrowSpeed`: The rate as an APR expressed as a decimal scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Set Base Minimum For Rewards

This function sets the minimum amount of base asset supplied to the protocol in order for accounts to accrue rewards.

Configurator

```
function setBaseMinForRewards(address cometProxy,
uint104 newBaseMinForRewards) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newBaseMinForRewards`: The amount of base asset scaled up by 10 to the “decimals” integer in the base asset’s contract.
- `RETURN`: No return, reverts on error.

Set Borrow Minimum

This function sets the minimum amount of base token that is allowed to be borrowed.

Configurator

```
function setBaseBorrowMin(address cometProxy, uint104
newBaseBorrowMin) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `setBaseBorrowMin`: The minimum borrow as an unsigned integer scaled up by 10 to the “decimals” integer in the base asset’s contract.
- `RETURN`: No return, reverts on error.

Set Target Reserves

This function sets the target reserves amount. Once the protocol reaches this amount of reserves of base asset, liquidators cannot buy collateral from the protocol.

Configurator

```
function setTargetReserves(address cometProxy, uint104
newTargetReserves) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newTargetReserves`: The amount of reserves of base asset as an unsigned integer scaled up by 10 to the “decimals” integer in the base asset’s contract.
- `RETURN`: No return, reverts on error.

Add a New Asset

This function adds an asset to the protocol through governance.

Configurator

```
function addAsset(address cometProxy, AssetConfig
calldata assetConfig) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `assetConfig`: The configuration that is added to the array of protocol asset configurations.
- `RETURN`: No return, reverts on error.

Update an Existing Asset

This function modifies an existing asset’s configuration parameters.

Configurator

```
function updateAsset(address cometProxy, AssetConfig
calldata newAssetConfig) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `newAssetConfig`: The configuration that is modified in the array of protocol asset configurations. All parameters are overwritten.
- `RETURN`: No return, reverts on error.

Update Asset Price Feed

This function updates the price feed contract address for a specific asset.

Configurator

```
function updateAssetPriceFeed(address cometProxy,
address asset, address newPriceFeed) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `asset`: The address of the underlying asset smart contract.
- `newPriceFeed`: The address of the new price feed smart contract.
- `RETURN`: No return, reverts on error.

Update Borrow Collateral Factor

This function updates the borrow collateral factor for an asset in the protocol.

Configurator

```
function updateAssetBorrowCollateralFactor(address
cometProxy, address asset, uint64 newBorrowCF)
external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `asset`: The address of the underlying asset smart contract.
- `newBorrowCF`: The collateral factor as an integer that represents the decimal value scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Update Liquidation Collateral Factor

This function updates the liquidation collateral factor for an asset in the protocol.

Configurator

```
function updateAssetLiquidateCollateralFactor(address
cometProxy, address asset, uint64 newLiquidateCF)
external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `asset`: The address of the underlying asset smart contract.
- `newLiquidateCF`: The collateral factor as an integer that represents the decimal value scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Update Liquidation Factor

This function updates the liquidation factor for an asset in the protocol.

The liquidation factor is a decimal value that is between 0 and 1 (inclusive) which determines the amount that is paid out to an underwater account upon liquidation.

The following is an example of the liquidation factor's role in a Compound III liquidation:

An underwater account has supplied $100 of WBTC as collateral. If the WBTC liquidation factor is `0.9`, the user will receive $90 of the base asset when a liquidator triggers an absorption of their account.

Configurator

```
function updateAssetLiquidationFactor(address
cometProxy, address asset, uint64
newLiquidationFactor) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `asset`: The address of the underlying asset smart contract.
- `newLiquidationFactor`: The factor as an integer that represents the decimal value scaled up by `10 ^ 18`.
- `RETURN`: No return, reverts on error.

Set Asset Supply Cap

This function sets the maximum amount of an asset that can be supplied to the protocol. Supply transactions will revert if the total supply would be greater than this number as a result.

Configurator

```
function updateAssetSupplyCap(address cometProxy,
address asset, uint128 newSupplyCap) external
```

- `cometProxy`: The address of the Comet proxy to set the configuration for.
- `asset`: The address of the underlying asset smart contract.
- `newSupplyCap`: The amount of the asset as an unsigned integer scaled up by 10 to the “decimals” integer in the asset’s contract.
- `RETURN`: No return, reverts on error.

ERC-20 Approve Manager Address

This function sets the Comet contract’s ERC-20 allowance of an asset for a manager address. It can only be called by the Governor.

In the event of a governance attack, an attacker could create a proposal that leverages this function to give themselves permissions to freely transfer all ERC-20 tokens out of the Comet contract.

Hypothetically, the attacker would need to either acquire supreme voting weight or add a malicious step in an otherwise innocuous and popular proposal and the community would fail to detect before approving.

Comet

```
function approveThis(address manager, address asset,
uint amount) override external
```

- `manager`: The address of a manager account that has its allowance modified.
- `asset`: The address of the asset's smart contract.
- `amount`: The amount of the asset approved for the manager expressed as an integer.
- `RETURN`: No return, reverts on error.

Transfer Governor

This function changes the address of the Configurator's Governor.

Configurator

```
function transferGovernor(address newGovernor)
external
```

- `newGovernor`: The address of the new Governor for Configurator.
- `RETURN`: No return, reverts on error.

Withdraw Reserves

This function allows governance to withdraw base token reserves from the protocol and send them to a specified address. Only the Governor address may call this function.

Comet

```
function withdrawReserves(address to, uint amount)
external
```

- `to`: The address of the recipient of the base asset tokens.
- `amount`: The amount of the base asset to send scaled up by 10 to the “decimals” integer in the base asset’s contract.
- `RETURN`: No return, reverts on error.

Contract Address