Compound III Documentation

Compiled from Web Sources

April 14, 2025

Contents

1 Introduction **4**
- 1.1 Introduction . . . 4
  - 1.1.1 Networks . . . 4
  - 1.1.2 Protocol Contracts . . . 5
- 1.2 Developer Resources . . . 6
- 1.3 Security . . . 7
  - 1.3.1 Audits . . . 7

2 Interest Rates **8**
- 2.1 Get Supply Rate . . . 8
  - 2.1.1 Comet . . . 8
  - 2.1.2 Solidity . . . 8
  - 2.1.3 Ethers.js v5.x . . . 8
- 2.2 Get Borrow Rate . . . 9
  - 2.2.1 Comet . . . 9
  - 2.2.2 Solidity . . . 9
  - 2.2.3 Ethers.js v5.x . . . 9
- 2.3 Get Utilization . . . 9
  - 2.3.1 Comet . . . 9
  - 2.3.2 Solidity . . . 10
  - 2.3.3 Ethers.js v5.x . . . 10

3 Collateral & Borrowing **11**
- 3.1 Supply . . . 11
  - 3.1.1 Comet . . . 11
  - 3.1.2 Solidity . . . 12
  - 3.1.3 Ethers.js v5.x . . . 12
- 3.2 Withdraw or Borrow . . . 12
  - 3.2.1 Comet . . . 12
  - 3.2.2 Solidity . . . 12
  - 3.2.3 Ethers.js v5.x . . . 13
- 3.3 Collateral Balance . . . 13
  - 3.3.1 Comet . . . 13
  - 3.3.2 Solidity . . . 13
  - 3.3.3 Ethers.js v5.x . . . 13
- 3.4 Borrow Collateralization . . . 13
  - 3.4.1 Comet . . . 13
  - 3.4.2 Solidity . . . 13
  - 3.4.3 Ethers.js v5.x . . . 13

3.5 Minimum Borrow Balance . . . 14
3.5.1 Comet . . . 14
3.5.2 Solidity . . . 14
3.5.3 Ethers.js v5.x . . . 14

4 Liquidation **15**
4.1 Liquidatable Accounts . . . 15
4.1.1 Comet . . . 15
4.1.2 Solidity . . . 15
4.1.3 Ethers.js v5.x . . . 15
4.2 Absorb . . . 15
4.2.1 Comet . . . 15
4.2.2 Solidity . . . 16
4.2.3 Ethers.js v5.x . . . 16
4.3 Buy Collateral . . . 16
4.3.1 Comet . . . 16
4.3.2 Solidity . . . 16
4.3.3 Ethers.js v5.x . . . 16
4.4 Ask Price . . . 16
4.4.1 Comet . . . 17
4.4.2 Solidity . . . 17
4.4.3 Ethers.js v5.x . . . 17
4.5 Liquidator Points . . . 17
4.5.1 Comet . . . 17
4.5.2 Solidity . . . 17
4.5.3 Ethers.js v5.x . . . 17
4.6 Reserves . . . 18
4.6.1 Get Base Asset Reserves . . . 18
4.6.2 Get Collateral Asset Reserves . . . 18
4.6.3 Target Reserves . . . 19

5 Account Management **20**
5.1 Allow . . . 20
5.1.1 Comet . . . 20
5.1.2 Solidity . . . 20
5.1.3 Ethers.js v5.x . . . 20
5.2 Allow By Signature . . . 20
5.2.1 Comet . . . 20
5.2.2 Solidity . . . 21
5.2.3 Ethers.js v5.x . . . 21
5.3 User Nonce . . . 21
5.3.1 Comet . . . 21
5.3.2 Solidity . . . 21
5.3.3 Ethers.js v5.x . . . 21
5.4 Version . . . 21
5.4.1 Comet . . . 21
5.4.2 Solidity . . . 22
5.4.3 Ethers.js v5.x . . . 22
5.5 Account Permissions . . . 22
5.5.1 Comet . . . 22
5.5.2 Solidity . . . 22
5.5.3 Ethers.js v5.x . . . 22

5.6 Transfer . . . 22
5.6.1 Comet . . . 22
5.6.2 Solidity . . . 23
5.6.3 Ethers.js v5.x . . . 23
5.7 Interfaces & ERC-20 Compatibility . . . 23

6 Protocol Rewards **24**
6.1 Reward Accrual Tracking . . . 24
6.1.1 Comet . . . 24
6.1.2 Solidity . . . 24
6.1.3 Ethers.js v5.x . . . 24
6.2 Get Reward Accrued . . . 24
6.2.1 Comet Rewards . . . 24
6.2.2 Solidity . . . 24
6.2.3 Ethers.js v5.x . . . 24
6.3 Claim Rewards . . . 25
6.3.1 Comet Rewards . . . 25
6.3.2 Solidity . . . 25
6.3.3 Ethers.js v5.x . . . 25

7 Governance **26**
7.1 Multi-chain Governance . . . 26
7.2 Set Comet Factory . . . 26
7.2.1 Configurator . . . 26

8 Helper Functions **27**
8.1 Bulk Actions . . . 27
8.1.1 Invoke . . . 27

1 Introduction

1.1 Introduction

CompoundIII is an EVM compatible protocol that enables supplying of crypto assets as collateral in order to borrow the *base asset*. Accounts can also earn interest by supplying the base asset to the protocol.

The initial deployment of Compound III is on Ethereum and the base asset is USDC.

Please join the #**development** room in the Compound community Discord server as well as the forums at comp.xyz; Compound Labs and members of the community look forward to helping you build an application on top of Compound III. Your questions help us improve, so please don't hesitate to ask if you can't find what you are looking for here.

For documentation of the Compound v2 Protocol, see docs.compound.finance/v2.

1.1.1 Networks

The network deployment artifacts with contract addresses are available in the Comet repository `deployments/` folder.

The v3 proxy is the only address to be used to interact with a Compound III instance. It is the first address listed in each of the tabs below. To generate the proper CometInterfaceABI (`CometInterface.sol`), compile the Comet project using `yarn compile`.

Ethereum Mainnet - USDC Base

Contract	Address
cUSDCv3	0xc3d688B...c3
cUSDCv3 Implementation	0x528c57A...E1b
cUSDCv3 Ext	0x2856173...5B0
Configurator	0x316f970...6E3
Configurator Implementation	0xcFC1fA6...F4F
Proxy Admin	0x1EC63B5...779
Comet Factory	0xa7F7De6...1b4
Rewards	0x1B0e765...a40
Bulker	0x74a81F8...0C3
Governor	0xc0Da029...529
Timelock	0x6d903f6...925
USDC	0xA0b8699...B48
cbBTC	0xcbB7C00...3Bf
COMP	0xc00e94C...888
LINK	0x5149107...6CA
tBTC	0x18084fb...a88
UNI	0x1f9840a...984
WBTC	0x2260FAC...599
WETH	0xC02aaA3...Cc2
wstETH	0x7f39C58...Ca0

Ethereum Mainnet - WETH Base

Contract	Address
cWETHv3	0xA17581A...E94
cWETHv3 Implementation	0x1a7E64b...50d
cWETHv3 Ext	0xe2C1F54...030
cbETH	0xBe98951...704
ezETH	0xbf5495E...110
osETH	0xf1C9acD...E38
rETH	0xae78736...393
rsETH	0xA1290d6...5A7
rswETH	0xFAe103D...6c0
weETH	0xCd5fE23...7ee
ETHx	0xA35b1B3...15b

Polygon Mainnet - USDC Base

Contract	Address
cUSDCv3	0xF25212E...445
WMATIC	0x0d500B1...270

1.1.2 Protocol Contracts

cUSDCv3

This is the main proxy contract for interacting with the first Compound III market. The address is fixed and independent from future upgrades to the market. It is an OpenZeppelinTransparentUpgradeablePr

cUSDCv3 Implementation

This is the implementation of the market logic contract, as deployed by the Comet Factory via the Configurator.

Do not interact with this contract directly; instead use the cUSDCv3 proxy address with the Comet Interface ABI.

cUSDCv3 Ext

This is an extension of the market logic contract which supports some auxiliary/independent interfaces for the protocol. This is used to add additional functionality without requiring contract space in the main protocol contract.

Do not interact with this contract directly; instead use the cUSDCv3 proxy address with the Comet Interface ABI.

Configurator

This is a proxy contract for the `configurator`, which is used to set and update parameters of a Comet proxy contract. The configurator deploys implementations of the Comet logic contract according to its configuration. This pattern allows significant gas savings for users of the protocol by 'constantizing' the parameters of the protocol.

Configurator Implementation

This is the implementation of the Configurator contract, which can also be upgraded to support unforeseen changes to the protocol.

Proxy Admin

This is the admin of the Comet and Configurator proxy contracts. It is a `ProxyAdmin` as recommended/implemented by OpenZeppelin according to their upgradeability pattern.

Comet Factory

This is the factory contract capable of producing instances of the Comet implementation/logic contract, and invoked by the Configurator.

Rewards

This is a rewards contract which can hold rewards tokens (e.g. COMP, WETH) and allows claiming rewards by users, according to the core protocol tracking indices.

Bulker

This is an external contract that is not integral to Comet's function. It allows accounts to bulk multiple operations into a single transaction. This is a useful contract for Compound III user interfaces. The following is an example of steps in a bulked transaction.

- Wrap Ether to WETH
- Supply WETH collateral
- Supply WBTC collateral
- Borrow USDC

In addition to supplying, borrowing, and wrapping, the bulker contract can also transfer collateral within the protocol and claim rewards.

1.2 Developer Resources

The following developer guides and code repositories serve as resources for community members building on Compound. They detail the protocol deployment process, construction of new features, and code examples for implementing external apps that depend on Compound III as infrastructure.

1. Compound III Developer FAQ
2. Scenarios, Migrations, and Workflows
3. Creating a Compound III Liquidator
4. Building a Comet Extension

1.3 Security

The security of the Compound protocol is our highest priority; our development team, alongside third-party auditors and consultants, has invested considerable effort to create a protocol that we believe is safe and dependable. All contract code and balances are publicly verifiable, and security researchers are eligible for a bug bounty for reporting undiscovered vulnerabilities.

We believe that size, visibility, and time are the true test for the security of a smart contract; please exercise caution, and make your own determination of security and suitability.

1.3.1 Audits

The Compound protocol has been reviewed & audited by OpenZeppelin and ChainSecurity.

1. Compound III Audit by OpenZeppelin
2. Compound III Security Audit by ChainSecurity

2 Interest Rates

Users with a positive balance of the base asset earn interest, denominated in the base asset, based on a supply rate model; users with a negative balance pay interest based on a borrow rate model. These are separate interest rate models, and set by governance.

The supply and borrow interest rates are a function of the utilization rate of the base asset. Each model includes a utilization rate "kink" - above this point the interest rate increases more rapidly. Interest accrues every second using the block timestamp.

Collateral assets do not earn or pay interest.

2.1 Get Supply Rate

This function returns the per second supply rate as the decimal representation of a percentage scaled up by 10^{18}. The formula for producing the supply rate is:

```
## If the Utilization is less than or equal to the Kink parameter

SupplyRate = supplyPerSecondInterestRateBase +
    supplyPerSecondInterestRateSlopeLow * utilization

## Else

SupplyRate = supplyPerSecondInterestRateBase +
    supplyPerSecondInterestRateSlopeLow * supplyKink +
    supplyPerSecondInterestRateSlopeHigh * (utilization - supplyKink)
```

To calculate the Compound III supply APR as a percentage, pass the current utilization to this function, and divide the result by 10^{18} and multiply by the approximate number of seconds in one year and scale up by 100.

```
Seconds Per Year = 60 * 60 * 24 * 365
Utilization = getUtilization()
Supply Rate = getSupplyRate(Utilization)
Supply APR = Supply Rate / (10 ^ 18) * Seconds Per Year * 100
```

2.1.1 Comet

```
function getSupplyRate(uint utilization) public view returns (uint64)
```

- `utilization`: The utilization at which to calculate the rate.
- `RETURNS`: The per second supply rate as the decimal representation of a percentage scaled up by `10 ^ 18`. E.g. `317100000` indicates, roughly, a 1% APR.

2.1.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint supplyRate = comet.getSupplyRate(0.8e18);
```

2.1.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const supplyRate = await comet.callStatic.getSupplyRate(0.8e18);
```

2.2 Get Borrow Rate

This function returns the per second borrow rate as the decimal representation of a percentage scaled up by 10^{18}. The formula for producing the borrow rate is:

```
## If the Utilization is less than or equal to the Kink parameter

BorrowRate = borrowPerSecondInterestRateBase +
    borrowPerSecondInterestRateSlopeLow * utilization

## Else

BorrowRate = borrowPerSecondInterestRateBase +
    borrowPerSecondInterestRateSlopeLow * borrowKink +
    borrowPerSecondInterestRateSlopeHigh * (utilization - borrowKink)
```

To calculate the Compound III borrow APR as a percentage, pass the current utilization to this function, and divide the result by 10^{18} and multiply by the approximate number of seconds in one year and scale up by 100.

```
Seconds Per Year = 60 * 60 * 24 * 365
Utilization = getUtilization()
Borrow Rate = getBorrowRate(Utilization)
Borrow APR = Borrow Rate / (10 ^ 18) * Seconds Per Year * 100
```

2.2.1 Comet

```
function getBorrowRate(uint utilization) public view returns (uint64)
```

- `utilization`: The utilization at which to calculate the rate.
- `RETURNS`: The per second borrow rate as the decimal representation of a percentage scaled up by `10 ^ 18`. E.g. `317100000` indicates, roughly, a 1% APR.

2.2.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint borrowRate = comet.getBorrowRate(0.8e18);
```

2.2.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const borrowRate = await comet.callStatic.getBorrowRate(0.8e18);
```

2.3 Get Utilization

This function returns the current protocol utilization of the base asset. The formula for producing the utilization is:

`Utilization = TotalBorrows / TotalSupply`

2.3.1 Comet

```
function getUtilization() public view returns (uint)
```

- `RETURNS`: The current protocol utilization percentage as a decimal, represented by an unsigned integer, scaled up by `10 ^ 18`. E.g. `1e17 or 100000000000000000` is 10% utilization.

2.3.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint utilization = comet.getUtilization(); // example: 10000000000000000 (1%)
```

2.3.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const utilization = await comet.callStatic.getUtilization();
```

3 Collateral & Borrowing

Users can add collateral assets to their account using the *supply* function (see section 3.1). Collateral can only be added if the market is below its *supplyCap* (see section **??**), which limits the protocol's risk exposure to collateral assets.

Each collateral asset increases the user's borrowing capacity, based on the asset's *borrowCollateralFactor* (see section **??**). The borrowing collateral factors are percentages that represent the portion of collateral value that can be borrowed.

For instance, if the borrow collateral factor for WBTC is 85%, an account can borrow up to 85% of the USD value of its supplied WBTC in the base asset. Collateral factors can be fetched using the *Get Asset Info By Address* function (see section **??**).

The base asset can be borrowed using the *withdraw* function (see section 3.2); the resulting borrow balance must meet the borrowing collateral factor requirements. If a borrowing account subsequently fails to meet the borrow collateral factor requirements, it cannot borrow additional assets until it supplies more collateral, or reduces its borrow balance using the supply function.

Account *balances* for the base token are signed integers. An account balance greater than zero indicates the base asset is supplied and a balance less than zero indicates the base asset is borrowed. *Note: Base token balances for assets with 18 decimals will start to overflow at a value of* $2^{103}/10^{18} \approx 10$ *trillion.*

Account balances are stored internally in Comet as *principal* values (also signed integers). The principal value, also referred to as the day-zero balance, is what an account balance at T_0 would have to be for it to be equal to the account balance today after accruing interest.

Global *indices* for supply and borrow are unsigned integers that increase over time to account for the interest accrued on each side. When an account interacts with the protocol, the indices are updated and saved. An account's present balance can be calculated using the current index with the following formulas.

```
Balance = Principal * BaseSupplyIndex [Principal > 0]
Balance = Principal * BaseBorrowIndex [Principal < 0]
```

3.1 Supply

The supply function transfers an asset to the protocol and adds it to the account's balance. This function can be used to **supply collateral, supply the base asset, or repay an open borrow** of the base asset.

If the base asset is supplied resulting in the account having a balance greater than zero, the base asset earns interest based on the current supply rate. Collateral assets that are supplied do not earn interest.

There are three separate methods to supply an asset to Compound III. The first is on behalf of the caller, the second is to a separate account, and the third is for a manager on behalf of an account.

Before supplying an asset to Compound III, the caller must first execute the asset's ERC-20 approve of the Comet contract.

3.1.1 Comet

```
function supply(address asset, uint amount)

function supplyTo(address dst, address asset, uint amount)
```

```
4
5 function supplyFrom(address from, address dst, address asset, uint amount)
```

- `asset`: The address of the asset's smart contract.
- `amount`: The amount of the asset to supply to Compound III expressed as an integer. A value of `MaxUint256` will repay all of the `dst`'s base borrow balance.
- `dst`: The address that is credited with the supplied asset within the protocol.
- `from`: The address to supply from. This account must first use the Allow method in order to allow the sender to transfer its tokens prior to calling Supply.
- `RETURN`: No return, reverts on error.

3.1.2 Solidity

```
1 Comet comet = Comet(0xCometAddress);
2 comet.supply(0xERC20Address, 1000000);
```

3.1.3 Ethers.js v5.x

```
1 const comet = new ethers.Contract(contractAddress, abiJson, provider);
2 await comet.supply(usdcAddress, 1000000);
```

3.2 Withdraw or Borrow

The withdraw method is used to **withdraw collateral** that is not currently supporting an open borrow. Withdraw is **also used to borrow the base asset** from the protocol if the account has supplied sufficient collateral. It can also be called from an allowed manager address.

Compound III implements a minimum borrow position size which can be found as `baseBorrowMin` in the protocol configuration (see section **??**). A withdraw transaction to borrow that results in the account's borrow size being less than the `baseBorrowMin` will revert.

3.2.1 Comet

```
1 function withdraw(address asset, uint amount)
2
3 function withdrawTo(address to, address asset, uint amount)
4
5 function withdrawFrom(address src, address to, address asset, uint amount)
```

- `asset`: The address of the asset that is being withdrawn or borrowed in the transaction.
- `amount`: The amount of the asset to withdraw or borrow. A value of `MaxUint256` will withdraw all of the `src`'s base balance.
- `to`: The address to send the withdrawn or borrowed asset.
- `src`: The address of the account to withdraw or borrow on behalf of. The `withdrawFrom` method can only be called by an allowed manager.
- `RETURN`: No return, reverts on error.

3.2.2 Solidity

```
1 Comet comet = Comet(0xCometAddress);
2 comet.withdraw(0xwbtcAddress, 100000000);
```

3.2.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.withdraw(usdcAddress, 100000000);
```

3.3 Collateral Balance

This function returns the current balance of a collateral asset for a specified account in the protocol.

3.3.1 Comet

```
function collateralBalanceOf(address account, address asset) external view
    returns (uint128)
```

- `account`: The address of the account in which to retrieve a collateral balance.
- `asset`: The address of the collateral asset smart contract.
- `RETURNS`: The balance of the collateral asset in the protocol for the specified account as an unsigned integer scaled up by 10 to the "decimals" integer in the asset's contract.

3.3.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint balance = comet.collateralBalanceOf(0xAccount, 0xUsdcAddress);
```

3.3.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const balance = await comet.callStatic.collateralBalanceOf('0xAccount', '0
    xUsdcAddress');
```

3.4 Borrow Collateralization

This function returns true if the account passed to it has non-negative liquidity based on the borrow collateral factors. This function returns false if an account does not have sufficient liquidity to increase its borrow position. A return value of false does not necessarily imply that the account is presently liquidatable (see *isLiquidatable* function in section **??**).

3.4.1 Comet

```
function isBorrowCollateralized(address account) public view returns (bool)
```

- `account`: The account to examine collateralization.
- `RETURNS`: Returns true if the account has enough liquidity for borrowing.

3.4.2 Solidity

```
Comet comet = Comet(0xCometAddress);
bool isCollateralized = comet.isBorrowCollateralized(0xAccount);
```

3.4.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const isCollateralized = await comet.callStatic.isBorrowCollateralized('0
    xAccount');
```

3.5 Minimum Borrow Balance

This function returns the minimum borrow balance allowed in the base asset. An account's initial borrow size must be equal to or greater than this value. Subsequent borrows may be of any size.

3.5.1 Comet

```
function baseBorrowMin() public view returns (uint256)
```

- `RETURNS`: The minimum borrow balance allowed by the protocol as an unsigned integer scaled up by 10 to the "decimals" integer in the base asset's contract.

3.5.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint baseBorrowMin = comet.baseBorrowMin();
```

3.5.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const baseBorrowMin = await comet.callStatic.baseBorrowMin();
```

4 Liquidation

Liquidation is determined by *liquidation collateral factors* (see section **??**), which are separate and higher than borrow collateral factors (used to determine initial borrowing capacity), which protects borrowers & the protocol by ensuring a price buffer for all new positions. These also enable governance to reduce borrow collateral factors without triggering the liquidation of existing positions.

When an account's borrow balance exceeds the limits set by liquidation collateral factors, it is eligible for liquidation. A liquidator (a bot, contract, or user) can call the *absorb* function (see section 4.2), which relinquishes ownership of the accounts collateral, and returns the value of the collateral, minus a penalty (*liquidationFactor*, see section **??**), to the user in the base asset. The liquidated user has no remaining debt, and typically, will have an excess (interest earning) balance of the base asset.

Each absorption is paid for by the protocol's reserves of the base asset. In return, the protocol receives the collateral assets. If the remaining reserves are less than a governance-set *target* (see section 4.6.3), liquidators are able to *buy* the collateral at a *discount* (see section 4.4) using the base asset, which increases the protocol's base asset reserves.

4.1 Liquidatable Accounts

This function returns true if the account passed to it has negative liquidity based on the liquidation collateral factor. A return value of true indicates that the account is presently liquidatable.

4.1.1 Comet

```
function isLiquidatable(address account) public view returns (bool)
```

- `account`: The account to examine liquidatability.
- `RETURNS`: Returns true if the account is presently able to be liquidated.

4.1.2 Solidity

```
Comet comet = Comet(0xCometAddress);
bool isLiquidatable = comet.isLiquidatable(0xAccount);
```

4.1.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const isLiquidatable = await comet.callStatic.isLiquidatable('0xAccount');
```

4.2 Absorb

This function can be called by any address to liquidate an underwater account. It transfers the account's debt to the protocol account, decreases cash reserves to repay the account's borrows, and adds the collateral to the protocol's own balance. The caller has the amount of gas spent noted. In the future, they could be compensated via governance.

4.2.1 Comet

```
function absorb(address absorber, address[] calldata accounts)
```

- `absorber`: The account that is issued liquidator points during successful execution.
- `accounts`: An array of underwater accounts that are to be liquidated.

- `RETURN`: No return, reverts on error.

4.2.2 Solidity

```
Comet comet = Comet(0xCometAddress);
comet.absorb(0xMyAddress, [ 0xUnderwaterAddress ]);
```

4.2.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.absorb('0xMyAddress', [ '0xUnderwaterAddress' ]);
```

4.3 Buy Collateral

This function allows any account to buy collateral from the protocol, at a discount from the Price Feed's price, using base tokens. A minimum collateral amount should be specified to indicate the maximum slippage acceptable for the buyer.

This function can be used after an account has been liquidated and there is collateral available to be purchased. Doing so increases protocol reserves. The amount of collateral available can be found by calling the *Collateral Balance* function (see section 3.3). The price of the collateral can be determined by using the *quoteCollateral* function (see section 4.4).

4.3.1 Comet

```
function buyCollateral(address asset, uint minAmount, uint baseAmount, address
    recipient) external
```

- `asset`: The address of the collateral asset.
- `minAmount`: The minimum amount of collateral tokens that are to be received by the buyer, scaled up by 10 to the "decimals" integer in the collateral asset's contract.
- `baseAmount`: The amount of base tokens used to buy collateral scaled up by 10 to the "decimals" integer in the base asset's contract.
- `recipient`: The address that receives the purchased collateral.
- `RETURN`: No return, reverts on error.

4.3.2 Solidity

```
Comet comet = Comet(0xCometAddress);
comet.buyCollateral(0xAssetAddress, 5e18, 5e18, 0xRecipient);
```

4.3.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.buyCollateral('0xAssetAddress', 5e18, 5e18, '0xRecipient');
```

4.4 Ask Price

In order to repay the borrows of absorbed accounts, the protocol needs to sell the seized collateral. The *Ask Price* is the price of the asset to be sold at a discount (configured by governance). This function uses the price returned by the protocol's price feed. The discount of the asset is derived from the `StoreFrontPriceFactor` and the asset's `LiquidationFactor` using the following formula.

```
DiscountFactor = StoreFrontPriceFactor * (1e18 - Asset.LiquidationFactor)
```

4.4.1 Comet

```
function quoteCollateral(address asset, uint baseAmount) public view returns (
    uint)
```

- `address`: The address of the asset which is being quoted.
- `amount`: The amount of the base asset used to purchase discounted collateral, as an integer, scaled up by 10 to the "decimals" integer in the base asset's contract.
- `RETURN`: The amount of collateral asset that can be purchased using the base asset, as an integer, scaled up by 10 to the "decimals" integer in the collateral asset's contract.

4.4.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint askPrice = comet.quoteCollateral(0xERC20Address, 10000000000);
```

4.4.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const askPrice = await comet.callStatic.quoteCollateral('0xERC20Address',
    1000000);
```

4.5 Liquidator Points

The protocol keeps track of the successful executions of absorb by tallying liquidator "points" and gas the liquidator has spent.

4.5.1 Comet

```
mapping(address => LiquidatorPoints) public liquidatorPoints;
```

- `address`: The address of the liquidator account.
- `RETURN`: A struct containing the stored data pertaining to the liquidator account.
- `numAbsorbs`: A Solidity `uint32` of the number of times absorb was successfully called.
- `numAbsorbed`: A Solidity `uint64` of the number of accounts successfully absorbed by the protocol as a result of the liquidators call to the absorb function.
- `approxSpend`: A Solidity `uint128` of the sum of all gas spent by the liquidator that has called the absorb function.

4.5.2 Solidity

```
Comet comet = Comet(0xCometAddress);
LiquidatorPoints pointsData = comet.liquidatorPoints(0xLiquidatorAddress);
```

4.5.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const [ numAbsorbs, numAbsorbed, approxSpend ] = await comet.callStatic.
    liquidatorPoints('0xLiquidatorAddress');
```

4.6 Reserves

Reserves are a balance of the base or collateral asset, stored internally in the protocol, which automatically protect users from bad debt. Reserves can also be withdrawn or used through the governance process.

Reserves are generated in two ways: the difference in interest paid by borrowers, and earned by suppliers of the base asset, accrue as reserves into the protocol. Second, the liquidation process uses, and can add to, protocol reserves based on the target reserve level set by governance.

4.6.1 Get Base Asset Reserves

This function returns the amount of protocol reserves for the base asset as an integer.

Comet

```
function getReserves() public view returns (int)
```

- `RETURNS`: The amount of base asset stored as reserves in the protocol as an unsigned integer scaled up by 10 to the "decimals" integer in the asset's contract.

Solidity

```
Comet comet = Comet(0xCometAddress);
uint reserves = comet.getReserves();
```

Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const reserves = await comet.callStatic.getReserves();
```

4.6.2 Get Collateral Asset Reserves

This function returns the amount of protocol reserves for the specified collateral asset as an integer.

Comet

```
function getCollateralReserves(address asset) public view returns (uint)
```

- `RETURNS`: The amount of collateral asset stored as reserves in the protocol as an unsigned integer scaled up by 10 to the "decimals" integer in the collateral asset's contract.

Solidity

```
Comet comet = Comet(0xCometAddress);
uint reserves = comet.getCollateralReserves(0xCollateralAsset);
```

Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const reserves = await comet.callStatic.getCollateralReserves(
    collateralAssetAddress);
```

4.6.3 Target Reserves

This immutable value represents the target amount of reserves of the base token. If the protocol holds greater than or equal to this amount of reserves, the *buyCollateral* function (see section 4.3) can no longer be successfully called.

Comet

```
function targetReserves() public view returns (uint)
```

- `RETURN`: The target reserve value of the base asset as an integer, scaled up by 10 to the "decimals" integer in the base asset's contract.

Solidity

```
Comet comet = Comet(0xCometAddress);
uint targetReserves = comet.targetReserves();
```

Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const targetReserves = await comet.callStatic.targetReserves();
```

5 Account Management

In addition to self-management, Compound III accounts can enable other addresses to have write permissions for their account. Account managers can withdraw or transfer collateral within the protocol on behalf of another account. This is possible only after an account has enabled permissions by using the *allow* function (see section 5.1).

5.1 Allow

Allow or disallow another address to withdraw or transfer on behalf of the sender's address.

5.1.1 Comet

```
function allow(address manager, bool isAllowed)
```

- `msg.sender`: The address of an account to allow or disallow a manager for.
- `manager`: The address of an account that becomes or will no longer be the manager of the owner.
- `isAllowed`: True to add the manager and false to remove the manager.
- `RETURN`: No return, reverts on error.

5.1.2 Solidity

```
Comet comet = Comet(0xCometAddress);
comet.allow(0xmanager, true);
```

5.1.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.allow(managerAddress, true);
```

5.2 Allow By Signature

This is a separate version of the allow function that enables submission using an EIP-712 offline signature. For more details on how to create an offline signature, review EIP-712.

5.2.1 Comet

```
function allowBySig(
  address owner,
  address manager,
  bool isAllowed_,
  uint256 nonce,
  uint256 expiry,
  uint8 v,
  bytes32 r,
  bytes32 s
) external
```

- `owner`: The address of an account to allow or disallow a manager for. The signatory must be the owner address.
- `manager`: The address of an account that becomes or will no longer be the manager of the owner.
- `isAllowed`: True to add the manager and false to remove the manager.

- `nonce`: The contract state required to match the signature. This can be retrieved from the contract's public `userNonce` mapping.
- `expiry`: The time at which the signature expires. 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.

5.2.2 Solidity

```
Comet comet = Comet(0xCometAddress);
comet.allowBySig(0xowner, 0xmanager, true, nonce, expiry, v, r, s);
```

5.2.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.allowBySig('0xowner', '0xmanager', true, nonce, expiry, v, r, s);
```

5.3 User Nonce

This gets the user nonce, like an EVM account nonce, which is used by `allowBySig`.

5.3.1 Comet

```
function userNonce(address) returns (uint)
```

- `address`: The address of the account in which to get a nonce.
- `RETURN`: An integer of the specified account's nonce.

5.3.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint nonce = comet.userNonce(0xAccount);
```

5.3.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const nonce = await comet.callStatic.userNonce('0xAccount');
```

5.4 Version

This gets the protocol version which is used by `allowBySig`.

5.4.1 Comet

```
function version() view returns (string memory)
```

- `RETURN`: A string of the protocol version number.

5.4.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint version = comet.version(); %Note:  Return type is string, variable should be string
```

5.4.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const version = await comet.callStatic.version();
```

5.5 Account Permissions

This function returns a boolean that indicates the status of an account's management address.

5.5.1 Comet

```
function hasPermission(address owner, address manager) public view returns (
    bool)
```

- `owner`: The address of an account that can be managed by another.
- `manager`: The address of the account that can have manager permissions over another.
- `RETURNS`: Returns true if the `manager` address is presently a manager of the `owner` address.

5.5.2 Solidity

```
Comet comet = Comet(0xCometAddress);
bool isManager = comet.hasPermission(0xOwner, 0xManager);
```

5.5.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const isManager = await comet.callStatic.hasPermission('0xOwner', '0xManager');
```

5.6 Transfer

This function is used to transfer an asset within the protocol to another address. A manager of an account is also able to perform a transfer on behalf of the account. Account balances change but the asset does not leave the protocol contract. The transfer will fail if it would make the account liquidatable.

There are two variants of the transfer function: `transfer` and `transferAsset`. The former conforms to the ERC-20 standard and transfers the base asset, while the latter requires specifying a specific asset to transfer.

5.6.1 Comet

```
function transfer(address dst, uint amount)
```

```
function transferFrom(address src, address dst, uint amount)
```

```
function transferAsset(address dst, address asset, uint amount)
```

```
function transferAssetFrom(address src, address dst, address asset, uint amount
    )
```

- `dst`: The address of an account that is the receiver in the transaction.

- `src`: The address of an account that is the sender of the asset in the transaction. This transfer method can only be called by an allowed manager.
- `asset`: The ERC-20 address of the asset that is being sent in the transaction.
- `amount`: The amount of the asset to transfer. A value of `MaxUint256` will transfer all of the `src`'s base balance.
- `RETURN`: No return, reverts on error.

5.6.2 Solidity

```
Comet comet = Comet(0xCometAddress);
comet.transfer(0xreceiver, 0xwbtcAddress, 100000000);
```

5.6.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
await comet.transfer(receiverAddress, usdcAddress, 100000000);
```

5.7 Interfaces & ERC-20 Compatibility

The Comet contract is a fully compatible ERC-20 wrapper for the base token. All of the interface methods of ERC-20 are externally exposed for accounts that supply or borrow. The **CometInterface.sol** contract file contains an example of a Solidity interface for the Comet contract.

6 Protocol Rewards

Compound III has a built-in system for tracking rewards for accounts that use the protocol. The full history of accrual of rewards are tracked for suppliers and borrowers of the base asset. The rewards can be any ERC-20 token. In order for rewards to accrue to Compound III accounts, the configuration's `baseMinForRewards` threshold for total supply of the base asset must be met.

6.1 Reward Accrual Tracking

The reward accrual is tracked in the Comet contract and rewards can be claimed by users from an external Comet Rewards contract. Rewards are accounted for with up to 6 decimals of precision.

6.1.1 Comet

```
function baseTrackingAccrued(address account) external view returns (uint64);
```

- `RETURNS`: Returns the amount of reward token accrued based on usage of the base asset within the protocol for the specified account, scaled up by 10^6.

6.1.2 Solidity

```
Comet comet = Comet(0xCometAddress);
uint64 accrued = comet.baseTrackingAccrued(0xAccount);
```

6.1.3 Ethers.js v5.x

```
const comet = new ethers.Contract(contractAddress, abiJson, provider);
const accrued = await comet.callStatic.baseTrackingAccrued('0xAccount');
```

6.2 Get Reward Accrued

The amount of reward token accrued but not yet claimed for an account can be fetched from the external Comet Rewards contract.

6.2.1 Comet Rewards

```
struct RewardOwed {
    address token;
    uint owed;
}

function getRewardOwed(address comet, address account) external returns (
    RewardOwed memory)
```

- `RETURNS`: Returns the amount of reward token accrued but not yet claimed, scaled up by 10 to the "decimals" integer in the reward token's contract.

6.2.2 Solidity

```
CometRewards rewards = CometRewards(0xRewardsAddress);
RewardOwed reward = rewards.getRewardOwed(0xCometAddress, 0xAccount);
```

6.2.3 Ethers.js v5.x

```
const rewards = new ethers.Contract(contractAddress, abiJson, provider);
const [ tokenAddress, amtOwed ] = await rewards.callStatic.getRewardOwed(
    cometAddress, accountAddress);
```

6.3 Claim Rewards

Any account can claim rewards for a specific account. Account owners and managers can also claim rewards to a specific address. The claim functions are available on the external Comet Rewards contract.

6.3.1 Comet Rewards

```
function claim(address comet, address src, bool shouldAccrue) external
```

```
function claimTo(address comet, address src, address to, bool shouldAccrue)
    external
```

- `comet`: The address of the Comet contract.
- `src`: The account in which to claim rewards.
- `to`: The account in which to transfer the claimed rewards.
- `shouldAccrue`: If true, the protocol will account for the rewards owed to the account as of the current block before transferring.
- `RETURN`: No return, reverts on error.

6.3.2 Solidity

```
CometRewards rewards = CometRewards(0xRewardsAddress);
rewards.claim(0xCometAddress, 0xAccount, true);
```

6.3.3 Ethers.js v5.x

```
const rewards = new ethers.Contract(contractAddress, abiJson, provider);
await rewards.claim(cometAddress, accountAddress, true);
```

7 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.

7.1 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.

7.2 Set Comet Factory

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

7.2.1 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.

8 Helper Functions

8.1 Bulk Actions

The Compound III codebase contains the source code of an external contract called *Bulker* that is designed to allow multiple Comet functions to be called in a single transaction.

Use cases of the Bulker contract include but are not limited to:

- Supplying of a collateral asset and borrowing of the base asset.
- Supplying or withdrawing of the native EVM token (like Ether) directly.
- Transferring or withdrawing of the base asset without leaving dust in the account.

8.1.1 Invoke

This function allows callers to pass an array of action codes and calldatas that are executed, one by one, in a single transaction.

Bulker

```
/// @notice The action for supplying an asset to Comet
bytes32 public constant ACTION_SUPPLY_ASSET = "ACTION_SUPPLY_ASSET";

/// @notice The action for supplying a native asset (e.g. ETH on Ethereum mainnet) to Comet
bytes32 public constant ACTION_SUPPLY_NATIVE_TOKEN = "ACTION_SUPPLY_NATIVE_TOKEN";

/// @notice The action for transferring an asset within Comet
bytes32 public constant ACTION_TRANSFER_ASSET = "ACTION_TRANSFER_ASSET";

/// @notice The action for withdrawing an asset from Comet
bytes32 public constant ACTION_WITHDRAW_ASSET = "ACTION_WITHDRAW_ASSET";

/// @notice The action for withdrawing a native asset from Comet
bytes32 public constant ACTION_WITHDRAW_NATIVE_TOKEN = "ACTION_WITHDRAW_NATIVE_TOKEN";

/// @notice The action for claiming rewards from the Comet rewards contract
bytes32 public constant ACTION_CLAIM_REWARD = "ACTION_CLAIM_REWARD";

function invoke(bytes32[] calldata actions, bytes[] calldata data) external payable
```

- `actions`: An array of bytes32 strings that correspond to the actions defined in the contract.
- `data`: An array of calldatas for each action to be called in the invoke transaction.
  - Supply Asset, Withdraw Asset, Transfer Asset
    - `comet`: The address of the Comet instance to interact with.
    - `to`: The destination address, within or external to the protocol.
    - `asset`: The address of the ERC-20 asset contract.
    - `amount`: The amount of the asset as an unsigned integer scaled up by 10 to the "decimals" integer in the asset's contract.
  - Supply Native, Withdraw Native (native chain token like ETH on Ethereum Mainnet)

    * `comet`: The address of the Comet instance to interact with.
    * `to`: The destination address, within or external to the protocol.
    * `amount`: The amount of the native token as an unsigned integer scaled up by 10 to the number of decimals of precision of the native EVM token.
- `RETURN`: No return, reverts on error.

Solidity

```
Bulker bulker = Bulker(0xBulkerAddress);
// ERC-20 ‘approve‘ the bulker. Then Comet ‘allow‘ the bulker to be a manager before calling ‘invoke‘.
bytes memory supplyAssetCalldata = (abi.encode('0xCometAddress', '0xAccount', '0xAsset', amount);
bulker.invoke([ 'ACTION_SUPPLY_ASSET' ], [ supplyAssetCalldata ]);
```

Ethers.js v5.x

```
const bulker = new ethers.Contract(contractAddress, abiJson, provider);
// ERC-20 ‘approve‘ the bulker. Then Comet ‘allow‘ the bulker to be a manager before calling ‘invoke‘.
const supplyAssetCalldata = ethers.utils.defaultAbiCoder.encode(['address', 'address', 'address', 'uint'], ['0xCometAddress', '0xAccount', '0xAsset', amount]);
await bulker.invoke([ 'ACTION_SUPPLY_ASSET' ], [ supplyAssetCalldata ]);
```