

# 智能合约审计报告

安全状态

**安全**



主测人： 知道创宇区块链安全研究团队

## 版本说明

修订内容	时间	修订者	版本号
编写文档	20210112	知道创宇区块链安全研究团队	V1.0

## 文档信息

文档名称	文档版本	报告编号	保密级别
FiIDA 智能合约审计报告	V1.0	b388c24fcd484223a926503b813d7 05f	项目组公开

## 声明

创宇仅就本报告出具前已经发生或存在的事实出具本报告，并就此承担相应责任。对于出具以后发生或存在的事实，创宇无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向创宇提供的文件和资料。创宇假设已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，创宇对由此而导致的损失和不利影响不承担任何责任。

# 目录

1. 综述.....	- 1 -
2. 代码漏洞分析.....	- 4 -
2.1 漏洞等级分布.....	- 4 -
2.2 审计结果汇总说明.....	- 5 -
3. 业务安全性检测.....	- 7 -
3.1. 配置功能【通过】 .....	- 7 -
3.2. 控制器功能【通过】 .....	- 8 -
3.3. 价格预言机【通过】 .....	- 11 -
3.4. 代币功能【通过】 .....	- 13 -
4. 代码基本漏洞检测.....	- 17 -
4.1. 编译器版本安全【通过】 .....	- 17 -
4.2. 冗余代码【通过】 .....	- 17 -
4.3. 安全算数库的使用【通过】 .....	- 17 -
4.4. 不推荐的编码方式【通过】 .....	- 17 -
4.5. require/assert 的合理使用【通过】 .....	- 18 -
4.6. fallback 函数安全【通过】 .....	- 18 -
4.7. tx.origin 身份验证【通过】 .....	- 18 -
4.8. owner 权限控制【通过】 .....	- 18 -
4.9. gas 消耗检测【通过】 .....	- 19 -
4.10. call 注入攻击【通过】 .....	- 19 -

4.11.	低级函数安全【通过】	- 19 -
4.12.	增发代币漏洞【通过】	- 19 -
4.13.	访问控制缺陷检测【通过】	- 20 -
4.14.	数值溢出检测【通过】	- 20 -
4.15.	算术精度误差【通过】	- 21 -
4.16.	错误使用随机数【通过】	- 21 -
4.17.	不安全的接口使用【通过】	- 21 -
4.18.	变量覆盖【通过】	- 22 -
4.19.	未初始化的储存指针【通过】	- 22 -
4.20.	返回值调用验证【通过】	- 22 -
4.21.	交易顺序依赖【通过】	- 23 -
4.22.	时间戳依赖攻击【通过】	- 23 -
4.23.	拒绝服务攻击【通过】	- 24 -
4.24.	假充值漏洞【通过】	- 24 -
4.25.	重入攻击检测【通过】	- 24 -
4.26.	重放攻击检测【通过】	- 25 -
4.27.	重排攻击检测【通过】	- 25 -
5.	附录 A: 合约代码	- 27 -
6.	附录 B: 安全风险评级标准	- 160 -
7.	附录 C: 智能合约安全审计工具简介	- 161 -
7.1.	Manticore	- 161 -
7.2.	Oyente	- 161 -

7.3.	securify.sh.....	- 161 -
7.4.	Echidna.....	- 161 -
7.5.	MAIAN.....	- 161 -
7.6.	ethersplay.....	- 162 -
7.7.	ida-evm.....	- 162 -
7.8.	Remix-ide.....	- 162 -
7.9.	知道创宇区块链安全审计人员专用工具包.....	- 162 -

## 1. 综述

本次报告有效测试时间是从 2021 年 1 月 5 日开始到 2021 年 1 月 12 日结束，在此期间针对 FiIDA 智能合约代码的安全性和规范性进行审计并以此作为报告统计依据。

此次测试中，知道创宇工程师对智能合约的常见漏洞（见第三章节）进行了全面的分析，综合评定为通过。

### 本次智能合约安全审计结果：通过

由于本次测试过程在非生产环境下进行，所有代码均为最新备份，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

本次审计的报告信息：

报告编号：b388c24fcd484223a926503b813d705f

报告查询地址链接：

<https://attest.im/attestation/searchResult?query=b388c24fcd484223a926503b813d705f>

本次测试的目标信息：

条目	描述
Token 名称	FiIDA
代码类型	代币代码、DeFi 协议代码、以太坊智能合约代码
代码语言	solidity

合约文件及哈希：

合约文件	MD5
SToken.sol	0adde19d810a413dd2c688c316a1262f
Qstroller.sol	961acfbc800df6bb7e2af173eb2d58d8

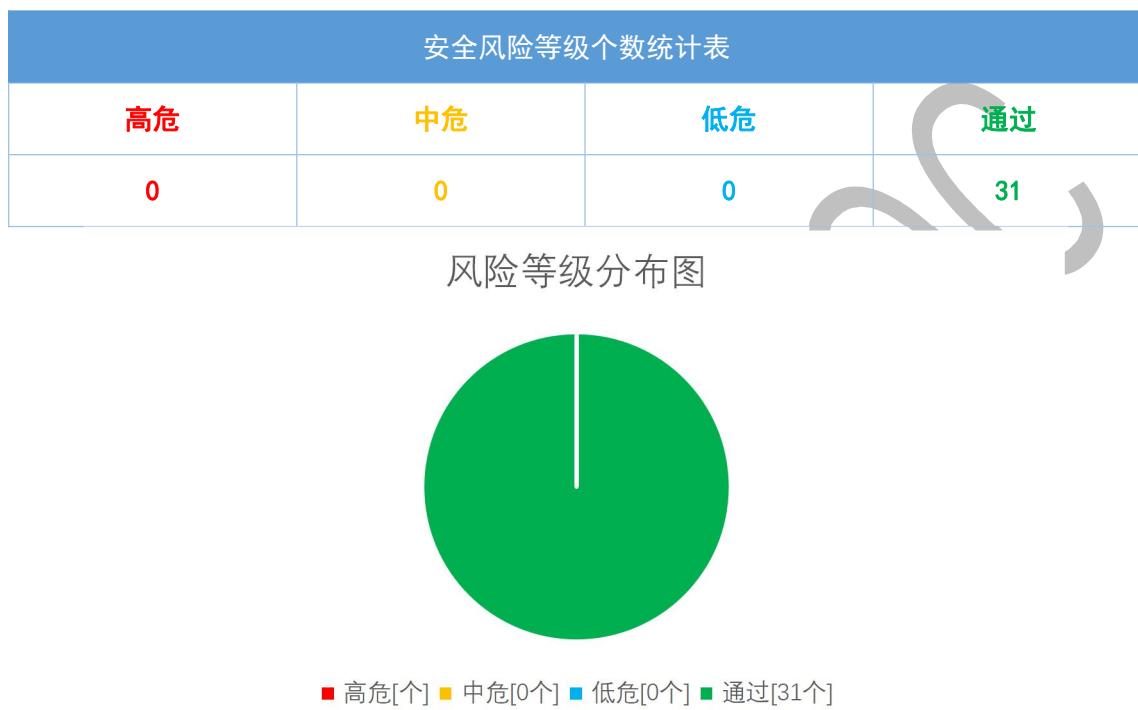
<code>QsPriceOracle.sol</code>	e6c1dd470f03dc08ce5667b894f921e5
<code>QsConfig.sol</code>	5ff27f0844106b202aa6edfb17c16aa
<code>Ownable.sol</code>	f881f173af570604003f3fb3d652f666
<code>IPriceCollector.sol</code>	ccfcfb1b35b4419f80346f4eb2788fa5
<code>Comp.sol</code>	707f728a3eb2bcd67f37ed5ed3c67244
<code>GovernorAlpha.sol</code>	aa6329672b8f078e017ff40cf4735501
<code>CompoundLens.sol</code>	29536534aa532fe995a8a1f864022222
<code>CarefulMath.sol</code>	b1f19e9f4ff15ac6673cd0d4ce242b01
<code>CDaiDelegate.sol</code>	b0cc054b06bb0680192c6ba2692402a9
<code>CErc20.sol</code>	28b1fd837b212644a245dcc8f6e7d0a4
<code>CErc20Delegate.sol</code>	f07e57da8e44d23d2d99737d27126e1d
<code>CErc20Delegator.sol</code>	6c470bc9603afffc0cc6c7be0d517a63a
<code>CErc20Immutable.sol</code>	f935ea442c903467e41ef569ba2cd4f0
<code>CEther.sol</code>	d2c11be85e4647509551a00d7f54f104
<code>Comptroller.sol</code>	08a181e1a7b1f68d7e623206e0285a4d
<code>ComptrollerG1.sol</code>	704501722442ff744e73972a1bd2adb2
<code>ComptrollerG2.sol</code>	465b11b04152b1b0796e1dfccb6d31e1
<code>ComptrollerG3.sol</code>	fb2ec2be2987950e17498505158129b3
<code>ComptrollerInterface.sol</code>	6cdb79a0889d8c37c8c5642d0813b37a
<code>ComptrollerStorage.sol</code>	c325b1d599b7d0d359b274c68b3ff0f2

CToken.sol	391d90a2d338738d4d54cfb0f31d372d
CTokenInterfaces.sol	c5c81b131f5b7bbf75d0c6f7128e8aaaf
DAIInterestRateModelV2.sol	74d4598de0d5d16aa22a6b9965492236
EIP20Interface.sol	fa93e469fb558e63a43784a83f62fc89
EIP20NonStandardInterface.sol	233b54ba1f055b8c2b9ea1c1dd3608f3
ErrorReporter.sol	4360952ef4d39b9c916546d941dca6d1
Exponential.sol	eaf3b583cd84e37d7a28223c88d8a114
InterestRateModel.sol	f8e83b5b683c7150fb53ea27df826cbe
JumpRateModel.sol	1f0d388f2ccdbda7b0926a6bc70db8e7
Maximillion.sol	ff32fc4d8bb119d9c6e8945d95774658
Migrations.sol	ca8d6ca8a6edf34f149a5095a8b074c9
PriceOracle.sol	3a863051264cbd4ef4f328bbf8bc5650
PriceOracleProxy.sol	45c08b15fcc3288c1b72423868dd9fdb
Reservoir.sol	113b5858eb4b59e5cd76bd651d9524c5
SafeMath.sol	51e3bce6f64c8cbdb9f6a6c4d8ccaa1b
SimplePriceOracle.sol	2c2c0b93265e2988d7450ceae6a5c8fd
Timelock.sol	32f77c167fae6e68ce3869f94a20775f
Unitroller.sol	481805e9c40a023d03e6d832a9ffd667
WhitePaperInterestRateModel.sol	4336abae2e23b69bb562af199d944b6e

## 2. 代码漏洞分析

### 2.1 漏洞等级分布

本次漏洞风险按等级统计：



## 2.2 审计结果汇总说明

审计结果			
审计项目	审计内容	状态	描述
业务安全性检测	配置合约	通过	经检测，不存在安全问题。
	控制器合约	通过	经检测，不存在安全问题。
	价格预言机	通过	经检测，不存在安全问题。
	代币合约	通过	经检测，不存在安全问题。
代码基本漏洞检测	编译器版本安全	通过	经检测，不存在该安全问题。
	冗余代码	通过	经检测，不存在该安全问题。
	安全算数库的使用	通过	经检测，不存在该安全问题。
	不推荐的编码方式	通过	经检测，不存在该安全问题。
	require/assert 的合理使用	通过	经检测，不存在该安全问题。
	fallback 函数安全	通过	经检测，不存在该安全问题。
	tx.origin 身份验证	通过	经检测，不存在该安全问题。
	owner 权限控制	通过	经检测，不存在该安全问题。
	gas 消耗检测	通过	经检测，不存在该安全问题。
	call 注入攻击	通过	经检测，不存在该安全问题。
	低级函数安全	通过	经检测，不存在该安全问题。
	增发代币漏洞	通过	经检测，不存在该安全问题。
	访问控制缺陷检测	通过	经检测，不存在该安全问题。
	数值溢出检测	通过	经检测，不存在该安全问题。
	算数精度误差	通过	经检测，不存在该安全问题。
	错误使用随机数检测	通过	经检测，不存在该安全问题。
	不安全的接口使用	通过	经检测，不存在该安全问题。
	变量覆盖	通过	经检测，不存在该安全问题。

	未初始化的存储指针	通过	经检测，不存在该安全问题。
	返回值调用验证	通过	经检测，不存在该安全问题。
	交易顺序依赖检测	通过	经检测，不存在该安全问题。
	时间戳依赖攻击	通过	经检测，不存在该安全问题。
	拒绝服务攻击检测	通过	经检测，不存在该安全问题。
	假充值漏洞检测	通过	经检测，不存在该安全问题。
	重入攻击检测	通过	经检测，不存在该安全问题。
	重放攻击检测	通过	经检测，不存在该安全问题。
	重排攻击检测	通过	经检测，不存在该安全问题。

### 3. 业务安全性检测

#### 3.1. 配置功能 【通过】

**审计分析：**配置功能在 QsConfig.sol 合约中实现，继承自 Ownable 和 Compound 项目的 Exponential 合约，主要用于管理员设置代币地址、金库地址、利率等关键配置。

```
contract QsConfig is Ownable, Exponential {  
    bool public compSpeedGuardianPaused = true;  
    address public compToken;  
    uint public safetyVaultRatio;  
    address public safetyVault;  
  
    event NewCompToken(address oldCompToken, address newCompToken);  
    event NewSafetyVault(address oldSafetyVault, address newSafetyVault);  
    event NewSafetyVaultRatio(uint oldSafetyVaultRatio, uint newSafetyVault);  
  
    function _setCompToken(address _compToken) public onlyOwner {  
        //knownsec// 设置 comp  
        //代币,仅 owner 调用  
        address oldCompToken = compToken;  
        compToken = _compToken;  
        emit NewCompToken(oldCompToken, compToken);  
    }  
  
    function _setSafetyVault(address _safetyVault) public onlyOwner {  
        //knownsec// 设置金库,仅  
        //owner 调用  
        address oldSafetyVault = safetyVault;  
        safetyVault = _safetyVault;  
        emit NewSafetyVault(oldSafetyVault, safetyVault);  
    }  
}
```

```

function _setSafetyVaultRatio(uint _safetyVaultRatio) public onlyOwner {//knownsec// 设置金库利率,仅 owner 调用
    uint oldSafetyVaultRatio = safetyVaultRatio;
    safetyVaultRatio = _safetyVaultRatio;
    emit NewSafetyVaultRatio(oldSafetyVaultRatio, safetyVaultRatio);
}

function _setCompSpeedGuardianPaused(bool state) public onlyOwner returns (bool)
//knownsec// 设置暂停状态,仅 owner 调用
    compSpeedGuardianPaused = state;
    return state;
}

function calculateSeizeTokenAllocation(uint _seizeTokenAmount, uint
liquidationIncentiveMantissa) public view returns(uint liquidatorAmount, uint safetyVaultAmount)
//knownsec// 计算代币分发量
    Exp memory vaultRatio = Exp({mantissa:safetyVaultRatio});
    (Exp memory tmp) = mulScalar(vaultRatio, _seizeTokenAmount);
    safetyVaultAmount = div_(tmp, liquidationIncentiveMantissa).mantissa;
    liquidatorAmount = sub_(_seizeTokenAmount, safetyVaultAmount);
}

```

**安全建议：**无。

### 3.2. 控制器功能【通过】

**审计分析：**控制器功能在 Qstroller.sol 合约中实现，继承自 Compound 的 Comptroller，主要用于调配出币速度、分发代币等。

```

contract Qstroller is Comptroller {
    QsConfig public qsConfig;
}

```

```
function _setQsConfig(QsConfig _qsConfig) public {//knownsec// 设置 QsConfig, 仅管理员调用
    require(msg.sender == admin, "Only admin can set quick silver configuration data.");
    qsConfig = _qsConfig;
}

/*
 * @notice Sets new governance token distribution speed
 * @dev Admin function to set new token distribution speed
 */

function _setCompSpeeds(address[] memory _allMarkets, uint[] memory _compSpeeds)
public {//knownsec// 设置出币速度
    // Check caller is admin
    require(msg.sender == admin, "Only admin can update token distribution");
    require(_allMarkets.length == _compSpeeds.length, "Incomplete parameter");
    require(allMarkets.length == _allMarkets.length, "Must update token distribution
within one transaction");

    uint _compRate = 0;
    for (uint i = 0; i < _allMarkets.length; i++) {
        address cToken = _allMarkets[i];
        Market storage market = markets[cToken];
        if (market.isComped == false) {
            _addCompMarketInternal(cToken);
        }
        compSpeeds[cToken] = _compSpeeds[i];
        _compRate = add(_compRate, _compSpeeds[i]);
    }
    _setCompRate(_compRate);
}
```

```
function refreshCompSpeeds() public {//knownsec// 刷新出币速度
    require(!qsConfig.compSpeedGuardianPaused(), "compSpeed is paused");
    require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");

    refreshCompSpeedsInternal();
}

function refreshCompSpeedsInternal() internal {
    if (qsConfig.compSpeedGuardianPaused()) {
        return;
    } else {
        super.refreshCompSpeedsInternal();
    }
}

function getCompAddress() public view returns (address) {//knownsec// 取代币地址
    return qsConfig.compToken();
}

function calculateSeizeTokenAllocation(uint _seizeTokenAmount) public view returns(uint liquidatorAmount, uint safetyVaultAmount) {
    return qsConfig.calculateSeizeTokenAllocation(_seizeTokenAmount,
liquidationIncentiveMantissa);
}

function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint)
//knownsec// 分发代币
{
    address compAddress = getCompAddress();
    if (userAccrued >= threshold && userAccrued > 0 && compAddress != address(0x0))
    {
        EIP20Interface comp = EIP20Interface(compAddress);
        uint compRemaining = comp.balanceOf(address(this));
    }
}
```

```
if (userAccrued <= compRemaining) {  
    comp.transfer(user, userAccrued);  
    return 0;  
}  
}  
return userAccrued;  
}  
}
```

安全建议：无。

### 3.3. 价格预言机 【通过】

**审计分析：** 价格预言机功能在 QsPriceOracle.sol 合约中实现，继承自 IPriceCollector 和 Compound 项目的 PriceOracle，主要用于价格管理员设置底层资产价格和各代币资产的直接价格。

```
contract QsPriceOracle is PriceOracle, IPriceCollector {  
    mapping(address => uint) prices;  
    address public priceAdmin;  
  
    event PricePosted(address asset, uint previousPriceMantissa, uint requestedPriceMantissa,  
        uint newPriceMantissa);  
    event PriceAdminTransferred(address indexed previousAdmin, address indexed newAdmin);  
  
    modifier onlyAdmin {  
        require(msg.sender == priceAdmin, "Price Admin required.");  
        _;  
    }  
  
    constructor() public {  
        priceAdmin = msg.sender;  
    }  
}
```

```
function getUnderlyingPrice(CToken cToken) public view returns (uint) {//knownsec// 获取底层资产价格
    if (SToken(address(cToken)).isNativeToken()) {
        return 1e18;
    } else {
        return prices[address(CErc20(address(cToken)).underlying())];
    }
}

function setUnderlyingPrice(CToken cToken, uint underlyingPriceMantissa) public onlyAdmin //knownsec// 设置底层资产价格,仅管理员调用
{
    address asset = address(CErc20(address(cToken)).underlying());
    setDirectPrice(asset, underlyingPriceMantissa);
}

function setDirectPrice(address _asset, uint _price) public onlyAdmin //knownsec// 设置直接价格,仅管理员调用
{
    prices[_asset] = _price;
    emit PricePosted(_asset, prices[_asset], _price, _price);
}

function setDirectPrice(address[] memory _assets, uint[] memory _prices) public onlyAdmin //knownsec// 批量设置直接价格,仅管理员调用
{
    require(_assets.length > 0, "At least one asset price is required");
    require(_assets.length == _prices.length, "Assets and prices are not match");

    for (uint i = 0; i < _assets.length; i++) {
        prices[_assets[i]] = _prices[i];
        emit PricePosted(_assets[i], prices[_assets[i]], _prices[i], _prices[i]);
    }
}
```

```
function assetPrices(address asset) external view returns (uint) {//knownsec// 获取资产价格
    return prices[asset];
}

function compareStrings(string memory a, string memory b) internal pure returns (bool)
//knownsec// 字符串比较
{
    return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
}

function transferPriceAdmin(address newAdmin) public onlyAdmin //knownsec// 转移管理员权限,仅管理员调用
{
    require(newAdmin != address(0), "Ownable: new price admin is the zero address");
    emit PriceAdminTransferred(priceAdmin, newAdmin);
    priceAdmin = newAdmin;
}
```

安全建议：无。

### 3.4. 代币功能【通过】

**审计分析：**代币功能在 SToken.sol 合约中实现，继承自 Compound 项目的 CToken，仅重写了 CToken 中原有的 seizeInternal 函数，该函数用于将市场的抵押代币转移至清算人。

```
contract SToken is CToken {

    function seizeInternal(address seizerToken, address liquidator, address borrower, uint seizeTokens) internal returns (uint) {
        /* Fail if seize not allowed */
        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, seizeTokens);
        if (allowed != 0) {
```

```
        return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
    }

    /* Fail if borrower = liquidator */
    if (borrower == liquidator) {
        return fail(Error.INVALID_ACCOUNT_PAIR,
FailureInfo.LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER);
    }

    MathError mathErr;
    uint borrowerTokensNew;
    uint liquidatorTokensNew;
    uint safetyVaultTokensNew;
    uint safetyVaultTokens;
    uint liquidatorSeizeTokens;

    (liquidatorSeizeTokens, safetyVaultTokens) =
Qstroller(address(comptroller)).qsConfig().calculateSeizeTokenAllocation(seizeTokens,
Qstroller(address(comptroller)).liquidationIncentiveMantissa());
    address safetyVault = Qstroller(address(comptroller)).qsConfig().safetyVault();

    /*
     * We calculate the new borrower and liquidator token balances, failing on
underflow/overflow:
     */
    * borrowerTokensNew = accountTokens[borrower] - seizeTokens
    * liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
    */

    (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED, uint(mathErr));
    }
```

```
(mathErr,      liquidatorTokensNew)      =      addUInt(accountTokens[liquidator],  
liquidatorSeizeTokens);  
  
if (mathErr != MathError.NO_ERROR) {  
  
    return                               failOpaque(Error.MATH_ERROR,  
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));  
}  
  
  
(mathErr,      safetyVaultTokensNew)      =      addUInt(accountTokens[safetyVault],  
safetyVaultTokens);  
  
if (mathErr != MathError.NO_ERROR) {  
  
    return                               failOpaque(Error.MATH_ERROR,  
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));  
}  
  
/////////////////////////////  
// EFFECTS & INTERACTIONS  
// (No safe failures beyond this point)  
  
/* We write the previously calculated values into storage */  
accountTokens[borrower] = borrowerTokensNew;  
accountTokens[liquidator] = liquidatorTokensNew;  
accountTokens[safetyVault] = safetyVaultTokensNew;  
  
/* Emit a Transfer event */  
emit Transfer(borrower, liquidator, liquidatorSeizeTokens);  
emit Transfer(borrower, safetyVault, safetyVaultTokens);  
  
/* We call the defense hook */  
comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower, seizeTokens);  
  
return uint(Error.NO_ERROR);  
}
```

```
function isNativeToken() public pure returns (bool) {  
    return false;  
}  
}
```

安全建议：无。

knownsec

## 4. 代码基本漏洞检测

### 4.1. 编译器版本安全 【通过】

检查合约代码实现中是否使用了安全的编译器版本

**检测结果:** 经检测, 智能合约代码中指定了编译器版本 0.5.16 以上, 不存在该安全问题。

**安全建议:** 无。

### 4.2. 冗余代码 【通过】

检查合约代码实现中是否包含冗余代码

**检测结果:** 经检测, 智能合约代码中不存在该安全问题。

**安全建议:** 无。

### 4.3. 安全算数库的使用 【通过】

检查合约代码实现中是否使用了 SafeMath 安全算数库

**检测结果:** 经检测, 智能合约代码中已使用 SafeMath 安全算数库, 不存在该安全问题。

**安全建议:** 无。

### 4.4. 不推荐的编码方式 【通过】

检查合约代码实现中是否有官方不推荐或弃用的编码方式

**检测结果:** 经检测, 智能合约代码中不存在该安全问题。

安全建议：无。

#### 4.5. require/assert 的合理使用 【通过】

检查合约代码实现中 require 和 assert 语句使用的合理性

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

#### 4.6. fallback 函数安全 【通过】

检查合约代码实现中是否正确使用 fallback 函数

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

#### 4.7. tx.origin 身份验证 【通过】

tx.origin 是 Solidity 的一个全局变量，它遍历整个调用栈并返回最初发送调用（或事务）的帐户的地址。在智能合约中使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

#### 4.8. owner 权限控制 【通过】

检查合约代码实现中的 owner 是否具有过高的权限。例如，任意修改其他账户余额等。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.9. gas 消耗检测 【通过】

检查 gas 的消耗是否超过区块最大限制

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.10. call 注入攻击 【通过】

call 函数调用时，应该做严格的权限控制，或直接写死 call 调用的函数。

**检测结果：**经检测，智能合约未使用 call 函数，不存在此漏洞。

**安全建议：**无。

#### 4.11. 低级函数安全 【通过】

检查合约代码实现中低级函数（call/delegatecall）的使用是否存在安全漏洞

call 函数的执行上下文是在被调用的合约中；而 delegatecall 函数的执行上下文是在当前调用该函数的合约中

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.12. 增发代币漏洞 【通过】

检查在初始化代币总量后，代币合约中是否存在可能使代币总量增加的函数。

**检测结果：**经检测，智能合约代码中不存在增发代币的功能。

**安全建议：**无。

#### 4.13. 访问控制缺陷检测 【通过】

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了 public、private 等关键词进行可见性修饰，

检查合约是否正确定义并使用了 modifier 对关键函数进行访问限制，避免越权导致的问题。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.14. 数值溢出检测 【通过】

智能合约中的算数问题是指数溢出和整数下溢。

Solidity 最多能处理 256 位的数字 ( $2^{256}-1$ )，最大数字增加 1 会溢出得到 0。同样，当数字为无符号类型时，0 减去 1 会下溢得到最大数字值。

整数溢出和下溢不是一种新类型的漏洞，但它们在智能合约中尤其危险。溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.15. 算术精度误差 【通过】

Solidity 作为一门编程语言具备和普通编程语言相似的数据结构设计，比如：变量、常量、函数、数组、函数、结构体等等，Solidity 和普通编程语言也有一个较大的区别——Solidity 没有浮点型，且 Solidity 所有的数值运算结果都只会是整数，不会出现小数的情况，同时也不允许定义小数类型数据。合约中的数值运算必不可少，而数值运算的设计有可能造成相对误差，例如同级运算： $5/2*10=20$ ，而  $5*10/2=25$ ，从而产生误差，在数据更大时产生的误差也会更大，更明显。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.16. 错误使用随机数 【通过】

智能合约中可能需要使用随机数，虽然 Solidity 提供的函数和变量可以访问明显难以预测的值，如 `block.number` 和 `block.timestamp`，但是它们通常或者比看起来更公开，或者受到矿工的影响，即这些随机数在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.17. 不安全的接口使用 【通过】

检查合约代码实现中是否使用了不安全的接口

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.18. 变量覆盖【通过】

检查合约代码实现中是否存在变量覆盖导致的安全问题

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.19. 未初始化的储存指针【通过】

在 solidity 中允许一个特殊的数据结构为 struct 结构体，而函数内的局部变量默认使用 storage 或 memory 储存。

而存在 storage(存储器)和 memory(内存)是两个不同的概念，solidity 允许指针指向一个未初始化的引用，而未初始化的局部 stroage 会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果，在开发中应该避免在函数中初始化 struct 变量。

**检测结果：**经检测，智能合约代码不使用结构体，不存在该问题。

**安全建议：**无。

## 4.20. 返回值调用验证【通过】

此问题多出现在和转币相关的智能合约中，故又称作静默失败发送或未经检查发送。

在 Solidity 中存在 transfer()、send()、call.value() 等转币方法，都可以用于向某一地址发送 Ether，其区别在于： transfer 发送失败时会 throw，并且进行状态回滚；只会传递 2300gas 供调用，防止重入攻击；send 发送失败时会返回 false；只会传递 2300gas 供调用，防止重入攻击；call.value 发送失败时会返回 false；

传递所有可用 gas 进行调用（可通过传入 gas\_value 参数进行限制），不能有效防止重入攻击。

如果在代码中没有检查以上 send 和 call.value 转币函数的返回值，合约会继续执行后面的代码，可能由于 Ether 发送失败而导致意外的结果。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.21. 交易顺序依赖【通过】

由于矿工总是通过代表外部拥有地址（EOA）的代码获取 gas 费用，因此用户可以指定更高的费用以便更快地开展交易。由于以太坊区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户提交了一个有价值的解决方案，恶意用户可以窃取该解决方案并以较高的费用复制其交易，以抢占原始解决方案。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无

#### 4.22. 时间戳依赖攻击【通过】

数据块的时间戳通常来说都是使用矿工的本地时间，而这个时间大约能有 900 秒的范围波动，当其他节点接受一个新区块时，只需要验证时间戳是否晚于之前的区块并且与本地时间误差在 900 秒以内。一个矿工可以通过设置区块的时间戳来尽可能满足有利于他的条件来从中获利。

检查合约代码实现中是否存在有依赖于时间戳的关键功能

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.23. 拒绝服务攻击【通过】

在以太坊的世界中，拒绝服务是致命的，遭受该类型攻击的智能合约可能永远无法恢复正常工作状态。导致智能合约拒绝服务的原因可能有很多种，包括在作为交易接收方时的恶意行为，人为增加计算功能所需 gas 导致 gas 耗尽，滥用访问控制访问智能合约的 private 组件，利用混淆和疏忽等等。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.24. 假充值漏洞【通过】

在代币合约的 transfer 函数对转账发起人(msg.sender)的余额检查用的是 if 判断方式，当 balances[msg.sender] < value 时进入 else 逻辑部分并 return false，最终没有抛出异常，我们认为仅 if/else 这种温和的判断方式在 transfer 这类敏感函数场景中是一种不严谨的编码方式。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

#### 4.25. 重入攻击检测【通过】

重入漏洞是最著名的以太坊智能合约漏洞，曾导致了以太坊的分叉（The DAO hack）。

Solidity 中的 call.value() 函数在被用来发送 Ether 的时候会消耗它接收到的所有 gas，当调用 call.value() 函数发送 Ether 的操作发生在实际减少发送者账户的余额之前时，就会存在重入攻击的风险。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 4.26. 重放攻击检测 【通过】

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击。在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。。

**检测结果：**经检测，智能合约未使用 call 函数，不存在此漏洞。

**安全建议：**无。

## 4.27. 重排攻击检测 【通过】

重排攻击是指矿工或其他方试图通过将自己的信息插入列表(list)或映射(mapping)中来与智能合约参与者进行“竞争”，从而使攻击者有机会将自己的信息存储到合约中。

**检测结果：**经检测，智能合约代码中不存在相关漏洞。

**安全建议：**无。

knownsec

## 5. 附录 A：合约代码

本次测试代码来源：

*SToken.sol*

```

pragma solidity ^0.5.16;

import "./compound/CToken.sol";
import "./Qstroller.sol!";

contract SToken is CToken {

    function seizeInternal(address seizerToken, address liquidator, address borrower, uint seizeTokens) internal
    returns (uint) {
        /* Fail if seize not allowed */
        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower, seizeTokens);
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return fail(Error.INVALID_ACCOUNT_PAIR,
FailureInfo.LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER);
        }

        MathError mathErr;
        uint borrowerTokensNew;
        uint liquidatorTokensNew;
        uint safetyVaultTokensNew;
        uint safetyVaultTokens;
        uint liquidatorSeizeTokens;

        (liquidatorSeizeTokens, safetyVaultTokens) =
Ostroller(address(comptroller)).qsConfig().calculateSeizeTokenAllocation(seizeTokens,
Qstroller(address(comptroller)).liquidationIncentiveMantissa());
        address safetyVault = Qstroller(address(comptroller)).qsConfig().safetyVault();
        /*
         * We calculate the new borrower and liquidator token balances, failing on underflow/overflow:
         *   borrowerTokensNew = accountTokens[borrower] - seizeTokens
         *   liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
         */
        (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED, uint(mathErr));
        }

        (mathErr, liquidatorTokensNew) = addUInt(accountTokens[liquidator], liquidatorSeizeTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
        }

        (mathErr, safetyVaultTokensNew) = addUInt(accountTokens[safetyVault], safetyVaultTokens);
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
        }

        /////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We write the previously calculated values into storage */
        accountTokens[borrower] = borrowerTokensNew;
        accountTokens[liquidator] = liquidatorTokensNew;
        accountTokens[safetyVault] = safetyVaultTokensNew;

        /* Emit a Transfer event */
        emit Transfer(borrower, liquidator, liquidatorSeizeTokens);
        emit Transfer(borrower, safetyVault, safetyVaultTokens);

        /* We call the defense hook */
        comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower, seizeTokens);
    }

    return uint(Error.NO_ERROR);
}

function isNativeToken() public pure returns (bool) {

```

```

        return false;
    }

}

Qstroller.sol
pragma solidity ^0.5.16;

import "./compound/Comptroller.sol";
import "./compound/EIP20Interface.sol";
import "./QsConfig.sol";

contract Qstroller is Comptroller {
    QsConfig public qsConfig;

    function _setQsConfig(QsConfig qsConfig) public {//knownsec// 设置QsConfig,仅管理员调用
        require(msg.sender == admin, "Only admin can set quick silver configuration data.");
        qsConfig = _qsConfig;
    }

    /**
     * @notice Sets new governance token distribution speed
     * @dev Admin function to set new token distribution speed
     */
    function _setCompSpeeds(address[] memory _allMarkets, uint[] memory _compSpeeds) public {//knownsec// 设置出币速度
        //Check caller is admin
        require(msg.sender == admin, "Only admin can update token distribution");
        require(_allMarkets.length == _compSpeeds.length, "Incomplete parameter");
        require(_allMarkets.length == _allMarkets.length, "Must update token distribution within one transaction");

        uint _compRate = 0;
        for (uint i = 0; i < _allMarkets.length; i++) {
            address cToken = _allMarkets[i];
            Market storage market = markets[cToken];
            if (market.isComped == false) {
                _addCompMarketInternal(cToken);
            }
            _compSpeeds[cToken] = _compSpeeds[i];
            _compRate = add(_compRate, _compSpeeds[i]);
        }
        _setCompRate(_compRate);
    }

    function refreshCompSpeeds() public {//knownsec// 刷新出币速度
        require(!qsConfig.compSpeedGuardianPaused(), "compSpeed is paused");
        require(msg.sender == tx.origin, "only externally owned accounts may refresh speeds");
        refreshCompSpeedsInternal();
    }

    function refreshCompSpeedsInternal() internal {
        if (qsConfig.compSpeedGuardianPaused()) {
            return;
        } else {
            super.refreshCompSpeedsInternal();
        }
    }

    function getCompAddress() public view returns (address) {//knownsec// 取代币地址
        return qsConfig.compToken();
    }

    function calculateSeizeTokenAllocation(uint _seizeTokenAmount) public view returns(uint liquidatorAmount, uint safetyVaultAmount) {
        return qsConfig.calculateSeizeTokenAllocation(_seizeTokenAmount, liquidationIncentiveMantissa);
    }

    function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {//knownsec// 分发代币
        address compAddress = getCompAddress();
        if(userAccrued >= threshold && userAccrued > 0 && compAddress != address(0x0)) {
            EIP20Interface comp = EIP20Interface(compAddress);
            uint compRemaining = comp.balanceOf(address(this));
            if(userAccrued <= compRemaining) {
                comp.transfer(user, userAccrued);
                return 0;
            }
        }
        return userAccrued;
    }
}

```

```

}

QsPriceOracle.sol
pragma solidity ^0.5.16;

import "./compound/PriceOracle.sol";
import "./compound/CErc20.sol";
import "./IPriceCollector.sol";
import "./SToken.sol";

contract QsPriceOracle is PriceOracle, IPriceCollector {
    mapping(address => uint) prices;
    address public priceAdmin;

    event PricePosted(address asset, uint previousPriceMantissa, uint requestedPriceMantissa, uint newPriceMantissa);
    event PriceAdminTransferred(address indexed previousAdmin, address indexed newAdmin);

    modifier onlyAdmin {
        require(msg.sender == priceAdmin, "Price Admin required.");
    }

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

    function getUnderlyingPrice(CToken cToken) public view returns (uint) {//knownsec// 获取底层资产价格
        if (SToken(address(cToken)).isNativeToken()) {
            return 1e18;
        } else {
            return prices[address(CErc20(address(cToken)).underlying())];
        }
    }

    function setUnderlyingPrice(CToken cToken, uint underlyingPriceMantissa) public onlyAdmin {//knownsec// 设置底层资产价格,仅管理员调用
        address asset = address(CErc20(address(cToken)).underlying());
        setDirectPrice(asset, underlyingPriceMantissa);
    }

    function setDirectPrice(address _asset, uint _price) public onlyAdmin {//knownsec// 设置直接价格,仅管理员调用
        prices[_asset] = _price;
        emit PricePosted(_asset, prices[_asset], _price, _price);
    }

    function setDirectPrice(address[] memory _assets, uint[] memory _prices) public onlyAdmin {//knownsec// 批量设置直接价格,仅管理员调用
        require(_assets.length > 0, "At least one asset price is required");
        require(_assets.length == _prices.length, "Assets and prices are not match");

        for (uint i = 0; i < _assets.length; i++) {
            prices[_assets[i]] = _prices[i];
            emit PricePosted(_assets[i], prices[_assets[i]], _prices[i], _prices[i]);
        }
    }

    function assetPrices(address asset) external view returns (uint) {//knownsec// 获取资产价格
        return prices[asset];
    }

    function compareStrings(string memory a, string memory b) internal pure returns (bool) {//knownsec// 字符串比较
        return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
    }

    function transferPriceAdmin(address newAdmin) public onlyAdmin {//knownsec// 转移管理员权限,仅管理员调用
        require(newAdmin != address(0), "Ownable: new price admin is the zero address");
        emit PriceAdminTransferred(priceAdmin, newAdmin);
        priceAdmin = newAdmin;
    }
}

QsConfig.sol
pragma solidity ^0.5.16;

import "./compound/Unitroller.sol";
import "./compound/Exponential.sol";
import "./Ownable.sol";

contract QsConfig is Ownable, Exponential {

```

```

bool public compSpeedGuardianPaused = true;
address public compToken;
uint public safetyVaultRatio;
address public safetyVault;

event NewCompToken(address oldCompToken, address newCompToken);
event NewSafetyVault(address oldSafetyVault, address newSafetyVault);
event NewSafetyVaultRatio(uint oldSafetyVaultRatio, uint newSafetyVault);

function _setCompToken(address _compToken) public onlyOwner {//knownsec// 设置 comp 代币, 仅 owner 调用
    address oldCompToken = compToken;
    compToken = _compToken;
    emit NewCompToken(oldCompToken, compToken);
}

function _setSafetyVault(address _safetyVault) public onlyOwner {//knownsec// 设置金库, 仅 owner 调用
    address oldSafetyVault = safetyVault;
    safetyVault = _safetyVault;
    emit NewSafetyVault(oldSafetyVault, safetyVault);
}

function _setSafetyVaultRatio(uint _safetyVaultRatio) public onlyOwner {//knownsec// 设置金库利率, 仅 owner 调用
    uint oldSafetyVaultRatio = safetyVaultRatio;
    safetyVaultRatio = _safetyVaultRatio;
    emit NewSafetyVaultRatio(oldSafetyVaultRatio, safetyVaultRatio);
}

function _setCompSpeedGuardianPaused(bool state) public onlyOwner returns (bool) {//knownsec// 设置暂停状态, 仅 owner 调用
    compSpeedGuardianPaused = state;
    return state;
}

function calculateSeizeTokenAllocation(uint seizeTokenAmount, uint liquidationIncentiveMantissa) public
view returns(uint liquidatorAmount, uint safetyVaultAmount) {//knownsec// 计算代币分发量
    Exp memory vaultRatio = Exp({mantissa:safetyVaultRatio});
    (Exp memory tmp) = mulScalar(vaultRatio, seizeTokenAmount);
    safetyVaultAmount = div_(tmp, liquidationIncentiveMantissa).mantissa;
    liquidatorAmount = sub_(seizeTokenAmount, safetyVaultAmount);
}
}

Ownable.sol
pragma solidity ^0.5.16;

contract Ownable {
    address public _owner;
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(_owner == msg.sender, "Ownable: caller is not the owner");
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

IPriceCollector.sol
pragma solidity ^0.5.16;

contract IPriceCollector {
    function setDirectPrice(address[] memory asset, uint[] memory price) public;
}

```

```
}
```

### Comp.sol

```
pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

contract Comp {
    ///@notice EIP-20 token name for this token
    string public constant name = "Compound";

    ///@notice EIP-20 token symbol for this token
    string public constant symbol = "Comp";

    ///@notice EIP-20 token decimals for this token
    uint8 public constant decimals = 18;

    ///@notice Total number of tokens in circulation
    uint public constant totalSupply = 10000000e18; // 10 million Comp

    ///@notice Allowance amounts on behalf of others
    mapping(address => mapping(address => uint96)) internal allowances;

    ///@notice Official record of token balances for each account
    mapping(address => uint96) internal balances;

    ///@notice A record of each accounts delegate
    mapping(address => address) public delegates;

    ///@notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }

    ///@notice A record of votes checkpoints for each account, by index
    mapping(address => mapping(uint32 => Checkpoint)) public checkpoints;

    ///@notice The number of checkpoints for each account
    mapping(address => uint32) public numCheckpoints;

    ///@notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");

    ///@notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

    ///@notice A record of states for signing / validating signatures
    mapping(address => uint) public nonces;

    ///@notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

    ///@notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

    ///@notice The standard EIP-20 transfer event
    event Transfer(address indexed from, address indexed to, uint256 amount);

    ///@notice The standard EIP-20 approval event
    event Approval(address indexed owner, address indexed spender, uint256 amount);

    /**
     * @notice Construct a new Comp token
     * @param account The initial account to grant all the tokens
     */
    constructor(address account) public {
        balances[account] = uint96(totalSupply);
        emit Transfer(address(0), account, totalSupply);
    }

    /**
     * @notice Get the number of tokens `spender` is approved to spend on behalf of `account`
     * @param account The address of the account holding the funds
     * @param spender The address of the account spending the funds
     * @return The number of tokens approved
     */
    function allowance(address account, address spender) external view returns (uint) {
        return allowances[account][spender];
    }

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     */
}
```

```


* @dev This will overwrite the approval amount for `spender`
* and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
* @param spender The address of the account which may transfer tokens
* @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
* @return Whether or not the approval succeeded
*/
function approve(address spender, uint rawAmount) external returns (bool) {
    uint96 amount;
    if (rawAmount == uint(-1)) {
        amount = uint96(-1);
    } else {
        amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");
    }
    allowances[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

/**
* @notice Get the number of tokens held by the `account`
* @param account The address of the account to get the balance of
* @return The number of tokens held
*/
function balanceOf(address account) external view returns (uint) {
    return balances[account];
}

/**
* @notice Transfer `amount` tokens from `msg.sender` to `dst`
* @param dst The address of the destination account
* @param rawAmount The number of tokens to transfer
* @return Whether or not the transfer succeeded
*/
function transfer(address dst, uint rawAmount) external returns (bool) {
    uint96 amount = safe96(rawAmount, "Comp::transfer: amount exceeds 96 bits");
    _transferTokens(msg.sender, dst, amount);
    return true;
}

/**
* @notice Transfer `amount` tokens from `src` to `dst`
* @param src The address of the source account
* @param dst The address of the destination account
* @param rawAmount The number of tokens to transfer
* @return Whether or not the transfer succeeded
*/
function transferFrom(address src, address dst, uint rawAmount) external returns (bool) {
    address spender = msg.sender;
    uint96 spenderAllowance = allowances[src][spender];
    uint96 amount = safe96(rawAmount, "Comp::approve: amount exceeds 96 bits");

    if (spender != src && spenderAllowance != uint96(-1)) {
        uint96 newAllowance = sub96(spenderAllowance, amount, "Comp::transferFrom: transfer amount exceeds spender allowance");
        allowances[src][spender] = newAllowance;
        emit Approval(src, spender, newAllowance);
    }

    _transferTokens(src, dst, amount);
    return true;
}

/**
* @notice Delegate votes from `msg.sender` to `delegatee`
* @param delegatee The address to delegate votes to
*/
function delegate(address delegatee) public {
    return _delegate(msg.sender, delegatee);
}

/**
* @notice Delegates votes from signatory to `delegatee`
* @param delegatee The address to delegate votes to
* @param nonce The contract state required to match the signature
* @param expiry The time at which to expire the signature
* @param v The recovery byte of the signature
* @param r Half of the ECDSA signature pair
* @param s Half of the ECDSA signature pair
*/
function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32 r, bytes32 s) public {
    bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)), getChainId(), address(this)));
    bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry));
    bytes32 digest = keccak256(abi.encodePacked("x19x01", domainSeparator, structHash));
}


```

```

address signatory = ecrecover(digest, v, r, s);
require(signatory != address(0), "Comp::delegateBySig: invalid signature");
require(nonce == nonces[signatory]++, "Comp::delegateBySig: invalid nonce");
require(now <= expiry, "Comp::delegateBySig: signature expired");
return _delegate(signatory, delegatee);
}

/**
 * @notice Gets the current votes balance for `account`
 * @param account The address to get votes balance
 * @return The number of current votes for `account`
 */
function getCurrentVotes(address account) external view returns (uint96) {
    uint32 nCheckpoints = numCheckpoints[account];
    return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
}

/**
 * @notice Determine the prior number of votes for an account as of a block number
 * @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
 * @param account The address of the account to check
 * @param blockNumber The block number to get the vote balance at
 * @return The number of votes the account had as of the given block
 */
function getPriorVotes(address account, uint blockNumber) public view returns (uint96) {
    require(blockNumber < block.number, "Comp::getPriorVotes: not yet determined");

    uint32 nCheckpoints = numCheckpoints[account];
    if(nCheckpoints == 0) {
        return 0;
    }

    // First check most recent balance
    if(checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
        return checkpoints[account][nCheckpoints - 1].votes;
    }

    // Next check implicit zero balance
    if(checkpoints[account][0].fromBlock > blockNumber) {
        return 0;
    }

    uint32 lower = 0;
    uint32 upper = nCheckpoints - 1;
    while (upper > lower) {
        uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
        Checkpoint memory cp = checkpoints[account][center];
        if(cp.fromBlock == blockNumber) {
            return cp.votes;
        } else if (cp.fromBlock < blockNumber) {
            lower = center;
        } else {
            upper = center - 1;
        }
    }
    return checkpoints[account][lower].votes;
}

function _delegate(address delegator, address delegatee) internal {
    address currentDelegate = delegates[delegator];
    uint96 delegatorBalance = balances[delegator];
    delegates[delegator] = delegatee;

    emit DelegateChanged(delegator, currentDelegate, delegatee);

    _moveDelegates(currentDelegate, delegatee, delegatorBalance);
}

function _transferTokens(address src, address dst, uint96 amount) internal {
    require(src != address(0), "Comp::_transferTokens: cannot transfer from the zero address");
    require(dst != address(0), "Comp::_transferTokens: cannot transfer to the zero address");

    balances[src] = sub96(balances[src], amount, "Comp::_transferTokens: transfer amount exceeds balance");
    balances[dst] = add96(balances[dst], amount, "Comp::_transferTokens: transfer amount overflows");
    emit Transfer(src, dst, amount);

    _moveDelegates(delegates[src], delegates[dst], amount);
}

function _moveDelegates(address srcRep, address dstRep, uint96 amount) internal {
    if(srcRep != dstRep && amount > 0) {
        if (srcRep != address(0)) {
            uint32 srcRepNum = numCheckpoints[srcRep];
            uint96 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
            uint96 srcRepNew = sub96(srcRepOld, amount, "Comp::_moveVotes: vote amount underflows");
            checkpoints[srcRep][srcRepNum - 1].votes = srcRepNew;
        }
        if (dstRep != address(0)) {
            uint32 dstRepNum = numCheckpoints[dstRep];
            uint96 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
            uint96 dstRepNew = add96(dstRepOld, amount, "Comp::_moveVotes: vote amount overflows");
            checkpoints[dstRep][dstRepNum - 1].votes = dstRepNew;
        }
    }
}

```

```

        _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
    }

    if (dstRep != address(0)) {
        uint32 dstRepNum = numCheckpoints[dstRep];
        uint96 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
        uint96 dstRepNew = add96(dstRepOld, amount, "Comp::_moveVotes: vote amount
overflows");
        _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
    }
}

function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint96 oldVotes, uint96 newVotes)
internal {
    uint32 blockNumber = safe32(block.number, "Comp::_writeCheckpoint: block number exceeds 32 bits");

    if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
        checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
    } else {
        checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
        numCheckpoints[delegatee] = nCheckpoints + 1;
    }

    emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
}

function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
    require(n < 2**32, errorMessage);
    return uint32(n);
}

function safe96(uint n, string memory errorMessage) internal pure returns (uint96) {
    require(n < 2**96, errorMessage);
    return uint96(n);
}

function add96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
    uint96 c = a + b;
    require(c >= a, errorMessage);
    return c;
}

function sub96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
    require(b <= a, errorMessage);
    return a - b;
}

function getChainId() internal pure returns (uint) {
    uint256 chainId;
    assembly { chainId := chainid() }
    return chainId;
}

GovernorAlpha.sol
pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

contract GovernorAlpha {
    ///@notice The name of this contract
    string public constant name = "Compound Governor Alpha";

    ///@notice The number of votes in support of a proposal required in order for a quorum to be reached and
    for a vote to succeed
    function quorumVotes() public pure returns (uint) { return 400000e18; } // 400,000 = 4% of Comp

    ///@notice The number of votes required in order for a voter to become a proposer
    function proposalThreshold() public pure returns (uint) { return 100000e18; } // 100,000 = 1% of Comp

    ///@notice The maximum number of actions that can be included in a proposal
    function proposalMaxOperations() public pure returns (uint) { return 10; } // 10 actions

    ///@notice The delay before voting on a proposal may take place, once proposed
    function votingDelay() public pure returns (uint) { return 1; } // 1 block

    ///@notice The duration of voting on a proposal, in blocks
    function votingPeriod() public pure returns (uint) { return 17280; } // ~3 days in blocks (assuming 15s blocks)

    ///@notice The address of the Compound Protocol Timelock
    TimelockInterface public timelock;

    ///@notice The address of the Compound governance token
    ComptrollerInterface public comp;
}

```

```

///@notice The address of the Governor Guardian
address public guardian;

///@notice The total number of proposals
uint public proposalCount;

struct Proposal {
    ///@notice Unique id for looking up a proposal
    uint id;

    ///@notice Creator of the proposal
    address proposer;

    ///@notice The timestamp that the proposal will be available for execution, set once the vote succeeds
    uint eta;

    ///@notice the ordered list of target addresses for calls to be made
    address[] targets;

    ///@notice The ordered list of values (i.e. msg.value) to be passed to the calls to be made
    uint[] values;

    ///@notice The ordered list of function signatures to be called
    string[] signatures;

    ///@notice The ordered list of calldata to be passed to each call
    bytes[] calldatas;

    ///@notice The block at which voting begins: holders must delegate their votes prior to this block
    uint startBlock;

    ///@notice The block at which voting ends: votes must be cast prior to this block
    uint endBlock;

    ///@notice Current number of votes in favor of this proposal
    uint forVotes;

    ///@notice Current number of votes in opposition to this proposal
    uint againstVotes;

    ///@notice Flag marking whether the proposal has been canceled
    bool canceled;

    ///@notice Flag marking whether the proposal has been executed
    bool executed;

    ///@notice Receipts of ballots for the entire set of voters
    mapping (address => Receipt) receipts;
}

///@notice Ballot receipt record for a voter
struct Receipt {
    ///@notice Whether or not a vote has been cast
    bool hasVoted;

    ///@notice Whether or not the voter supports the proposal
    bool support;

    ///@notice The number of votes the voter had, which were cast
    uint96 votes;
}

///@notice Possible states that a proposal may be in
enum ProposalState {
    Pending,
    Active,
    Canceled,
    Defeated,
    Succeeded,
    Queued,
    Expired,
    Executed
}

///@notice The official record of all proposals ever proposed
mapping (uint => Proposal) public proposals;

///@notice The latest proposal for each proposer
mapping (address => uint) public latestProposalIds;

///@notice The EIP-712 typehash for the contract's domain
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");

///@notice The EIP-712 typehash for the ballot struct used by the contract
bytes32 public constant BALLOT_TYPEHASH = keccak256("Ballot(uint256 proposalId,bool support)");

```

```

///@notice An event emitted when a new proposal is created
event ProposalCreated(uint id, address proposer, address[] targets, uint[] values, string[] signatures, bytes[] calldatas, uint startBlock, uint endBlock, string description);

///@notice An event emitted when a vote has been cast on a proposal
event VoteCast(address voter, uint proposalId, bool support, uint votes);

///@notice An event emitted when a proposal has been canceled
event ProposalCanceled(uint id);

///@notice An event emitted when a proposal has been queued in the Timelock
event ProposalQueued(uint id, uint eta);

///@notice An event emitted when a proposal has been executed in the Timelock
event ProposalExecuted(uint id);

constructor(address timelock_, address comp_, address guardian_) public {
    timelock = TimelockInterface(timelock_);
    comp = CompInterface(comp_);
    guardian = guardian_;
}

function propose(address[] memory targets, uint[] memory values, string[] memory signatures, bytes[] memory calldatas, string memory description) public returns (uint) {
    require(comp.getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold");
    require(targets.length == values.length && targets.length == signatures.length && targets.length == calldatas.length, "GovernorAlpha::propose: proposal function information arity mismatch");
    require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
    require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too many actions");

    uint latestProposalId = latestProposalIds[msg.sender];
    if (latestProposalId != 0) {
        ProposalState proposersLatestProposalState = state(latestProposalId);
        require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::propose: one live proposal per proposer, found an already active proposal");
        require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha::propose: one live proposal per proposer, found an already pending proposal");
    }

    uint startBlock = add256(block.number, votingDelay());
    uint endBlock = add256(startBlock, votingPeriod());

    proposalCount++;
    Proposal memory newProposal = Proposal({
        id: proposalCount,
        proposer: msg.sender,
        eta: 0,
        targets: targets,
        values: values,
        signatures: signatures,
        calldatas: calldatas,
        startBlock: startBlock,
        endBlock: endBlock,
        forVotes: 0,
        againstVotes: 0,
        canceled: false,
        executed: false
    });

    proposals[newProposal.id] = newProposal;
    latestProposalIds[newProposal.proposer] = newProposal.id;

    emit ProposalCreated(newProposal.id, msg.sender, targets, values, signatures, calldatas, startBlock, endBlock, description);
    return newProposal.id;
}

function queue(uint proposalId) public {
    require(state(proposalId) == ProposalState.Succeeded, "GovernorAlpha::queue: proposal can only be queued if it is succeeded");
    Proposal storage proposal = proposals[proposalId];
    uint eta = add256(block.timestamp, timelock.delay());
    for (uint i = 0; i < proposal.targets.length; i++) {
        queueOrRevert(proposal.targets[i], proposal.values[i], proposal.signatures[i], proposal.calldatas[i], eta);
    }
    proposal.eta = eta;
    emit ProposalQueued(proposalId, eta);
}

function _queueOrRevert(address target, uint value, string memory signature, bytes memory data, uint eta)
internal {
    require(!timelock.queuedTransactions(keccak256(abi.encode(target, value, signature, data, eta))), "GovernorAlpha::queueOrRevert: proposal action already queued at eta");
    timelock.queueTransaction(target, value, signature, data, eta);
}

```

```

function execute(uint proposalId) public payable {
    require(state(proposalId) == ProposalState.Queued, "GovernorAlpha::execute: proposal can only be
executed if it is queued");
    Proposal storage proposal = proposals[proposalId];
    proposal.executed = true;
    for (uint i = 0; i < proposal.targets.length; i++) {
        timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i], proposal.values[i],
proposal.signatures[i], proposal.calldatas[i], proposal.eta);
    }
    emit ProposalExecuted(proposalId);
}

function cancel(uint proposalId) public {
    ProposalState state = state(proposalId);
    require(state != ProposalState.Executed, "GovernorAlpha::cancel: cannot cancel executed proposal");

    Proposal storage proposal = proposals[proposalId];
    require(msg.sender == guardian || comp.getPriorVotes(proposal.proposer, sub256(block.number, 1)) <
proposalThreshold(), "GovernorAlpha::cancel: proposer above threshold");

    proposal.canceled = true;
    for (uint i = 0; i < proposal.targets.length; i++) {
        timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal.signatures[i],
proposal.calldatas[i], proposal.eta);
    }
    emit ProposalCanceled(proposalId);
}

function getActions(uint proposalId) public view returns (address[] memory targets, uint[] memory values,
string[] memory signatures, bytes[] memory calldatas) {
    Proposal storage p = proposals[proposalId];
    return (p.targets, p.values, p.signatures, p.calldatas);
}

function getReceipt(uint proposalId, address voter) public view returns (Receipt memory) {
    return proposals[proposalId].receipts[voter];
}

function state(uint proposalId) public view returns (ProposalState) {
    require(proposalCount >= proposalId && proposalId > 0, "GovernorAlpha::state: invalid proposal
id");
    Proposal storage proposal = proposals[proposalId];
    if (proposal.canceled) {
        return ProposalState.Canceled;
    } else if (block.number <= proposal.startBlock) {
        return ProposalState.Pending;
    } else if (block.number <= proposal.endBlock) {
        return ProposalState.Active;
    } else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes < quorumVotes()) {
        return ProposalState.Deleted;
    } else if (proposal.eta == 0) {
        return ProposalState.Succeeded;
    } else if (proposal.executed) {
        return ProposalState.Executed;
    } else if (block.timestamp >= add256(proposal.eta, timelock.GRACE_PERIOD())) {
        return ProposalState.Expired;
    } else {
        return ProposalState.Queued;
    }
}

function castVote(uint proposalId, bool support) public {
    return _castVote(msg.sender, proposalId, support);
}

function castVoteBySig(uint proposalId, bool support, uint8 v, bytes32 r, bytes32 s) public {
    bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)),
getChainId(), address(this)));
    bytes32 structHash = keccak256(abi.encode(BALLOT_TYPEHASH, proposalId, support));
    bytes32 digest = keccak256(abi.encodePacked("0x1901", domainSeparator, structHash));
    address signatory = ecrecover(digest, v, r, s);
    require(signatory != address(0), "GovernorAlpha::castVoteBySig: invalid signature");
    return _castVote(signatory, proposalId, support);
}

function _castVote(address voter, uint proposalId, bool support) internal {
    require(state(proposalId) == ProposalState.Active, "GovernorAlpha::_castVote: voting is closed");
    Proposal storage proposal = proposals[proposalId];
    Receipt storage receipt = proposal.receipts[voter];
    require(receipt.hasVoted == false, "GovernorAlpha::_castVote: voter already voted");
    uint96 votes = comp.getPriorVotes(voter, proposal.startBlock);

    if (support) {
        proposal.forVotes = add256(proposal.forVotes, votes);
    } else {
}

```

```

        proposal.againstVotes = add256(proposal.againstVotes, votes);
    }

    receipt.hasVoted = true;
    receipt.support = support;
    receipt.votes = votes;

    emit VoteCast(voter, proposalId, support, votes);
}

function __acceptAdmin() public {
    require(msg.sender == guardian, "GovernorAlpha::__acceptAdmin: sender must be gov guardian");
    timelock.acceptAdmin();
}

function __abdicate() public {
    require(msg.sender == guardian, "GovernorAlpha::__abdicate: sender must be gov guardian");
    guardian = address(0);
}

function __queueSetTimelockPendingAdmin(address newPendingAdmin, uint eta) public {
    require(msg.sender == guardian, "GovernorAlpha::__queueSetTimelockPendingAdmin: sender must be gov guardian");
    timelock.queueTransaction(address(timelock), 0, "setPendingAdmin(address)");
    abi.encode(newPendingAdmin, eta);
}

function __executeSetTimelockPendingAdmin(address newPendingAdmin, uint eta) public {
    require(msg.sender == guardian, "GovernorAlpha::__executeSetTimelockPendingAdmin: sender must be gov guardian");
    timelock.executeTransaction(address(timelock), 0, "setPendingAdmin(address)");
    abi.encode(newPendingAdmin, eta);
}

function add256(uint256 a, uint256 b) internal pure returns (uint) {
    uint c = a + b;
    require(c >= a, "addition overflow");
    return c;
}

function sub256(uint256 a, uint256 b) internal pure returns (uint) {
    require(b <= a, "subtraction underflow");
    return a - b;
}

function getChainId() internal pure returns (uint) {
    uint chainId;
    assembly {chainId := chainid()}
    return chainId;
}

interface TimelockInterface {
    function delay() external view returns (uint);
    function GRACE_PERIOD() external view returns (uint);
    function acceptAdmin() external;
    function queuedTransactions(bytes32 hash) external view returns (bool);
    function queueTransaction(address target, uint value, string calldata signature, bytes calldata data, uint eta)
    external returns (bytes32);
    function cancelTransaction(address target, uint value, string calldata signature, bytes calldata data, uint eta)
    external;
    function executeTransaction(address target, uint value, string calldata signature, bytes calldata data, uint eta)
    external payable returns (bytes memory);
}

interface CompInterface {
    function getPriorVotes(address account, uint blockNumber) external view returns (uint96);
}

CompoundLens.sol

pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

import "./CErc20.sol";
import "./CToken.sol";
import "./PriceOracle.sol";
import "./EIP20Interface.sol";
import "./Governance/GovernorAlpha.sol";
import "./Governance/Comp.sol";

interface ComptrollerLensInterface {
    function markets(address) external view returns (bool, uint);
    function oracle() external view returns (PriceOracle);
    function getAccountLiquidity(address) external view returns (uint, uint, uint);
    function getAssetsIn(address) external view returns (CToken[] memory);
}

```

```

function claimComp(address) external;
function compAccrued(address) external view returns (uint);
}

contract CompoundLens {
    struct CTokenMetadata {
        address cToken;
        uint exchangeRateCurrent;
        uint supplyRatePerBlock;
        uint borrowRatePerBlock;
        uint reserveFactorMantissa;
        uint totalBorrows;
        uint totalReserves;
        uint totalSupply;
        uint totalCash;
        bool isListed;
        uint collateralFactorMantissa;
        address underlyingAssetAddress;
        uint cTokenDecimals;
        uint underlyingDecimals;
    }

    function cTokenMetadata(CToken cToken) public returns (CTokenMetadata memory) {
        uint exchangeRateCurrent = cToken.exchangeRateCurrent();
        ComptrollerLensInterface comptroller = ComptrollerLensInterface(address(cToken.comptroller()));
        (bool isListed, uint collateralFactorMantissa) = comptroller.markets(address(cToken));
        address underlyingAssetAddress;
        uint underlyingDecimals;

        if (compareStrings(cToken.symbol(), "cETH")) {
            underlyingAssetAddress = address(0);
            underlyingDecimals = 18;
        } else {
            CErc20 cErc20 = CErc20(address(cToken));
            underlyingAssetAddress = cErc20.underlying();
            underlyingDecimals = EIP20Interface(cErc20.underlying()).decimals();
        }

        return CTokenMetadata({
            cToken: address(cToken),
            exchangeRateCurrent: exchangeRateCurrent,
            supplyRatePerBlock: cToken.supplyRatePerBlock(),
            borrowRatePerBlock: cToken.borrowRatePerBlock(),
            reserveFactorMantissa: cToken.reserveFactorMantissa(),
            totalBorrows: cToken.totalBorrows(),
            totalReserves: cToken.totalReserves(),
            totalSupply: cToken.totalSupply(),
            totalCash: cToken.getCash(),
            isListed: isListed,
            collateralFactorMantissa: collateralFactorMantissa,
            underlyingAssetAddress: underlyingAssetAddress,
            cTokenDecimals: cToken.decimals(),
            underlyingDecimals: underlyingDecimals
        });
    }

    function cTokenMetadataAll(CToken[] calldata cTokens) external returns (CTokenMetadata[] memory) {
        uint cTokenCount = cTokens.length;
        CTokenMetadata[] memory res = new CTokenMetadata[](cTokenCount);
        for (uint i = 0; i < cTokenCount; i++) {
            res[i] = cTokenMetadata(cTokens[i]);
        }
        return res;
    }

    struct CTokenBalances {
        address cToken;
        uint balanceOf;
        uint borrowBalanceCurrent;
        uint balanceOfUnderlying;
        uint tokenBalance;
        uint tokenAllowance;
    }

    function cTokenBalances(CToken cToken, address payable account) public returns (CTokenBalances memory) {
        uint balanceOf = cToken.balanceOf(account);
        uint borrowBalanceCurrent = cToken.borrowBalanceCurrent(account);
        uint balanceOfUnderlying = cToken.balanceOfUnderlying(account);
        uint tokenBalance;
        uint tokenAllowance;

        if (compareStrings(cToken.symbol(), "cETH")) {
            tokenBalance = account.balance;
            tokenAllowance = account.balance;
        } else {
            CErc20 cErc20 = CErc20(address(cToken));
        }
    }
}

```

```
EIP20Interface underlying = EIP20Interface(cErc20.underlying());
tokenBalance = underlying.balanceOf(account);
tokenAllowance = underlying.allowance(account, address(cToken));
}

return CTokenBalances({
    cToken: address(cToken),
    balanceOf: balanceOf,
    borrowBalanceCurrent: borrowBalanceCurrent,
    balanceOfUnderlying: balanceOfUnderlying,
    tokenBalance: tokenBalance,
    tokenAllowance: tokenAllowance
});
}

function cTokenBalancesAll(CToken[] calldata cTokens, address payable account) external returns
(CTokenBalances[] memory) {
    uint cTokenCount = cTokens.length;
    CTokenBalances[] memory res = new CTokenBalances[](cTokenCount);
    for (uint i = 0; i < cTokenCount; i++) {
        res[i] = cTokenBalances(cTokens[i], account);
    }
    return res;
}

struct CTokenUnderlyingPrice {
    address cToken;
    uint underlyingPrice;
}

function cTokenUnderlyingPrice(CToken cToken) public view returns (CTokenUnderlyingPrice memory) {
    ComptrollerLensInterface comptroller = ComptrollerLensInterface(address(cToken.comptroller()));
    PriceOracle priceOracle = comptroller.oracle();
    return CTokenUnderlyingPrice({
        cToken: address(cToken),
        underlyingPrice: priceOracle.getUnderlyingPrice(cToken)
    });
}

function cTokenUnderlyingPriceAll(CToken[] calldata cTokens) external view returns
(CTokenUnderlyingPrice[] memory) {
    uint cTokenCount = cTokens.length;
    CTokenUnderlyingPrice[] memory res = new CTokenUnderlyingPrice[](cTokenCount);
    for (uint i = 0; i < cTokenCount; i++) {
        res[i] = cTokenUnderlyingPrice(cTokens[i]);
    }
    return res;
}

struct AccountLimits {
    CToken[] markets;
    uint liquidity;
    uint shortfall;
}

function getAccountLimits(ComptrollerLensInterface comptroller, address account) public view returns
(AccountLimits memory) {
    (uint errorCode, uint liquidity, uint shortfall) = comptroller.getAccountLiquidity(account);
    require(errorCode == 0);

    return AccountLimits({
        markets: comptroller.getAssetsIn(account),
        liquidity: liquidity,
        shortfall: shortfall
    });
}

struct GovReceipt {
    uint proposalId;
    bool hasVoted;
    bool support;
    uint96 votes;
}

function getGovReceipts(GovernorAlpha governor, address voter, uint[] memory proposalIds) public view
returns (GovReceipt[] memory) {
    uint proposalCount = proposalIds.length;
    GovReceipt[] memory res = new GovReceipt[](proposalCount);
    for (uint i = 0; i < proposalCount; i++) {
        GovernorAlpha.Receipt memory receipt = governor.getReceipt(proposalIds[i], voter);
        res[i] = GovReceipt({
            proposalId: proposalIds[i],
            hasVoted: receipt.hasVoted,
            support: receipt.support,
            votes: receipt.votes
        });
    }
}
```

```
        }
        return res;
    }

    struct GovProposal {
        uint proposalId;
        address proposer;
        uint eta;
        address[] targets;
        uint[] values;
        string[] signatures;
        bytes[] calldatas;
        uint startBlock;
        uint endBlock;
        uint forVotes;
        uint againstVotes;
        bool canceled;
        bool executed;
    }

    function setProposal(GovProposal memory res, GovernorAlpha governor, uint proposalId) internal view {
        (
            address proposer,
            uint eta,
            uint startBlock,
            uint endBlock,
            uint forVotes,
            uint againstVotes,
            bool canceled,
            bool executed
        ) = governor.proposals(proposalId);
        res.proposalId = proposalId;
        res.proposer = proposer;
        res.eta = eta;
        res.startBlock = startBlock;
        res.endBlock = endBlock;
        res.forVotes = forVotes;
        res.againstVotes = againstVotes;
        res.canceled = canceled;
        res.executed = executed;
    }

    function getGovProposals(GovernorAlpha governor, uint[] calldata proposalIds) external view returns
    (GovProposal[] memory) {
        GovProposal[] memory res = new GovProposal[](proposalIds.length);
        for (uint i = 0; i < proposalIds.length; i++) {
            (
                address[] memory targets,
                uint[] memory values,
                string[] memory signatures,
                bytes[] memory calldatas
            ) = governor.getActions(proposalIds[i]);
            res[i] = GovProposal({
                proposalId: 0,
                proposer: address(0),
                eta: 0,
                targets: targets,
                values: values,
                signatures: signatures,
                calldatas: calldatas,
                startBlock: 0,
                endBlock: 0,
                forVotes: 0,
                againstVotes: 0,
                canceled: false,
                executed: false
            });
            setProposal(res[i], governor, proposalIds[i]);
        }
        return res;
    }

    struct CompBalanceMetadata {
        uint balance;
        uint votes;
        address delegate;
    }

    function getCompBalanceMetadata(Comp comp, address account) external view returns
    (CompBalanceMetadata memory) {
        return CompBalanceMetadata({
            balance: comp.balanceOf(account),
            votes: uint256(comp.getCurrentVotes(account)),
            delegate: comp.delegates(account)
        });
    }
```

```

struct CompBalanceMetadataExt {
    uint balance;
    uint votes;
    address delegate;
    uint allocated;
}

function getCompBalanceMetadataExt(Comp comp, ComptrollerLensInterface comptroller, address account)
external returns (CompBalanceMetadataExt memory) {
    uint balance = comp.balanceOf(account);
    comptroller.claimComp(account);
    uint newBalance = comp.balanceOf(account);
    uint accrued = comptroller.compAccrued(account);
    uint total = add(accrued, newBalance, "sum comp total");
    uint allocated = sub(total, balance, "sub allocated");

    return CompBalanceMetadataExt({
        balance: balance,
        votes: uint256(comp.getCurrentVotes(account)),
        delegate: comp.delegates(account),
        allocated: allocated
    });
}

function getCompBalanceWithAccrued(Comp comp, ComptrollerLensInterface comptroller, address account)
external returns (uint balance, uint allocated) {
    balance = comp.balanceOf(account);
    comptroller.claimComp(account);
    uint newBalance = comp.balanceOf(account);
    uint accrued = comptroller.compAccrued(account);
    uint total = add(accrued, newBalance, "sum comp total");
    allocated = sub(total, balance, "sub allocated");
}

struct CompVotes {
    uint blockNumber;
    uint votes;
}

function getCompVotes(Comp comp, address account, uint32[] calldata blockNumbers) external view returns
(CompVotes[] memory) {
    CompVotes[] memory res = new CompVotes[](blockNumbers.length);
    for (uint i = 0; i < blockNumbers.length; i++) {
        res[i] = CompVotes({
            blockNumber: uint256(blockNumbers[i]),
            votes: uint256(comp.getPriorVotes(account, blockNumbers[i]))
        });
    }
    return res;
}

function compareStrings(string memory a, string memory b) internal pure returns (bool) {
    return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
}

function add(uint a, uint b, string memory errorMessage) internal pure returns (uint) {
    uint c = a + b;
    require(c >= a, errorMessage);
    return c;
}

function sub(uint a, uint b, string memory errorMessage) internal pure returns (uint) {
    require(b <= a, errorMessage);
    uint c = a - b;
    return c;
}

```

### CarefulMath.sol

```

pragma solidity ^0.5.16;

/**
 * @title Careful Math
 * @author Compound
 * @notice Derived from OpenZeppelin's SafeMath library
 *         https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol
 */
contract CarefulMath {

    /**
     * @dev Possible error codes that we can return
     */
    enum MathError {
        NO_ERROR,

```

```

    DIVISION_BY_ZERO,
    INTEGER_OVERFLOW,
    INTEGER_UNDERFLOW
}

/**
 * @dev Multiplies two numbers, returns an error on overflow.
 */
function mulUInt(uint a, uint b) internal pure returns (MathError, uint) {
    if(a == 0) {
        return (MathError.NO_ERROR, 0);
    }
    uint c = a * b;
    if(c / a != b) {
        return (MathError.INTEGER_OVERFLOW, 0);
    } else {
        return (MathError.NO_ERROR, c);
    }
}

/**
 * @dev Integer division of two numbers, truncating the quotient.
 */
function divUInt(uint a, uint b) internal pure returns (MathError, uint) {
    if(b == 0) {
        return (MathError.DIVISION_BY_ZERO, 0);
    }
    return (MathError.NO_ERROR, a / b);
}

/**
 * @dev Subtracts two numbers, returns an error on overflow (i.e. if subtrahend is greater than minuend).
 */
function subUInt(uint a, uint b) internal pure returns (MathError, uint) {
    if(b <= a) {
        return (MathError.NO_ERROR, a - b);
    } else {
        return (MathError.INTEGER_UNDERFLOW, 0);
    }
}

/**
 * @dev Adds two numbers, returns an error on overflow.
 */
function addUInt(uint a, uint b) internal pure returns (MathError, uint) {
    uint c = a + b;
    if(c >= a) {
        return (MathError.NO_ERROR, c);
    } else {
        return (MathError.INTEGER_OVERFLOW, 0);
    }
}

/**
 * @dev add a and b and then subtract c
 */
function addThenSubUInt(uint a, uint b, uint c) internal pure returns (MathError, uint) {
    (MathError err0, uint sum) = addUInt(a, b);
    if(err0 != MathError.NO_ERROR) {
        return (err0, 0);
    }
    return subUInt(sum, c);
}

```

### CDaiDelegate.sol

```

pragma solidity ^0.5.16;

import "./CErc20Delegate.sol";

/**
 * @title Compound's CDai Contract
 * @notice CToken which wraps Multi-Collateral DAI
 * @author Compound
 */
contract CDaiDelegate is CErc20Delegate {
    /**
     * @notice DAI adapter address
    */

```

```

address public daiJoinAddress;
< /**
 * @notice DAI Savings Rate (DSR) pot address
 */
address public potAddress;
< /**
 * @notice DAI vat address
 */
address public vatAddress;
< /**
 * @notice Delegate interface to become the implementation
 * @param data The encoded arguments for becoming
 */
function _becomeImplementation(bytes memory data) public {
    require(msg.sender == admin, "only the admin may initialize the implementation");
    (address daiJoinAddress_, address potAddress_) = abi.decode(data, (address, address));
    return _becomeImplementation(daiJoinAddress_, potAddress_);
}

< /**
 * @notice Explicit interface to become the implementation
 * @param daiJoinAddress_ DAI adapter address
 * @param potAddress_ DAI Savings Rate (DSR) pot address
 */
function _becomeImplementation(address daiJoinAddress_, address potAddress_) internal {
    // Get dai and vat and sanity check the underlying
    DaiJoinLike daiJoin = DaiJoinLike(daiJoinAddress_);
    PotLike pot = PotLike(potAddress_);
    GemLike dai = daiJoin.dai();
    VatLike vat = daiJoin.vat();
    require(address(dai) == underlying, "DAI must be the same as underlying");

    // Remember the relevant addresses
    daiJoinAddress = daiJoinAddress_;
    potAddress = potAddress_;
    vatAddress = address(vat);

    // Approve moving our DAI into the vat through daiJoin
    dai.approve(daiJoinAddress, uint(-1));

    // Approve the pot to transfer our funds within the vat
    vat.hope(potAddress);
    vat.hope(daiJoinAddress);

    // Accumulate DSR interest -- must do this in order to doTransferIn
    pot.drip();

    // Transfer all cash in (doTransferIn does this regardless of amount)
    doTransferIn(address(this), 0);
}

< /**
 * @notice Delegate interface to resign the implementation
 */
function _resignImplementation() public {
    require(msg.sender == admin, "only the admin may abandon the implementation");

    // Transfer all cash out of the DSR - note that this relies on self-transfer
    DaiJoinLike daiJoin = DaiJoinLike(daiJoinAddress);
    PotLike pot = PotLike(potAddress);
    VatLike vat = VatLike(vatAddress);

    // Accumulate interest
    pot.drip();

    // Calculate the total amount in the pot, and move it out
    uint pie = pot.pie(address(this));
    pot.exit(pie);

    // Checks the actual balance of DAI in the vat after the pot exit
    uint bal = vat.dai(address(this));

    // Remove our whole balance
    daiJoin.exit(address(this), bal / RAY);
}

< *** CToken Overrides ***
< /**
 * @notice Accrues DSR then applies accrued interest to total borrows and reserves
 * @dev This calculates interest accrued from the last checkpointed block
 *      up to the current block and writes new checkpoint to storage.
 */

```

```

function accrueInterest() public returns (uint) {
    // Accumulate DSR interest
    PotLike(potAddress).drip();

    // Accumulate CToken interest
    return super.accrueInterest();
}

/***
 * @notice Gets balance of this contract in terms of the underlying
 * @dev This excludes the value of the current message, if any
 * @return The quantity of underlying tokens owned by this contract
*/
function getCashPrior() internal view returns (uint) {
    PotLike pot = PotLike(potAddress);
    uint pie = pot.pie(address(this));
    return mul(pot.chi(), pie) / RAY;
}

/***
 * @notice Transfer the underlying to this contract and sweep into DSR pot
 * @param from Address to transfer funds from
 * @param amount Amount of underlying to transfer
 * @return The actual amount that is transferred
*/
function doTransferIn(address from, uint amount) internal returns (uint) {
    // Perform the EIP-20 transfer in
    EIP20Interface token = EIP20Interface(underlying);
    require(token.transferFrom(from, address(this), amount), "unexpected EIP-20 transfer in return");

    DaiJoinLike daiJoin = DaiJoinLike(daiJoinAddress);
    GemLike dai = GemLike(underlying);
    PotLike pot = PotLike(potAddress);
    VatLike vat = VatLike(vatAddress);

    // Convert all our DAI to internal DAI in the vat
    daiJoin.join(address(this), dai.balanceOf(address(this)));

    // Checks the actual balance of DAI in the vat after the join
    uint bal = vat.dai(address(this));

    // Calculate the percentage increase to th pot for the entire vat, and move it in
    // Note: We may leave a tiny bit of DAI in the vat...but we do the whole thing every time
    uint pie = bal / pot.chi();
    pot.join(pie);

    return amount;
}

/***
 * @notice Transfer the underlying from this contract, after sweeping out of DSR pot
 * @param to Address to transfer funds to
 * @param amount Amount of underlying to transfer
*/
function doTransferOut(address payable to, uint amount) internal {
    DaiJoinLike daiJoin = DaiJoinLike(daiJoinAddress);
    PotLike pot = PotLike(potAddress);

    // Calculate the percentage decrease from the pot, and move that much out
    // Note: Use a slightly larger pie size to ensure that we get at least amount in the vat
    uint pie = add(mul(amount, RAY) / pot.chi(), 1);
    pot.exit(pie);

    daiJoin.exit(to, amount);
}

/***
 * @notice Maker Internals
*/
uint256 constant RAY = 10 ** 27;

function add(uint x, uint y) internal pure returns (uint z) {
    require((z = x + y) >= x, "add-overflow");
}

function mul(uint x, uint y) internal pure returns (uint z) {
    require(y == 0 || (z = x * y) / y == x, "mul-overflow");
}

/***
 * @notice Maker Interfaces
*/
interface PotLike {
    function chi() external view returns (uint);
    function pie(address) external view returns (uint);
    function drip() external returns (uint);
}

```

```
function join(uint) external;
function exit(uint) external;
}

interface GemLike {
    function approve(address, uint) external;
    function balanceOf(address) external view returns (uint);
    function transferFrom(address, address, uint) external returns (bool);
}

interface VatLike {
    function dai(address) external view returns (uint);
    function hope(address) external;
}

interface DaiJoinLike {
    function vat() external returns (VatLike);
    function dai() external returns (GemLike);
    function join(address, uint) external payable;
    function exit(address, uint) external;
}

CERC20.sol
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./SToken.sol";

<**
 * @title Compound's CErc20 Contract
 * @notice CTokens which wrap an EIP-20 underlying
 * @author Compound
 */
contract CErc20 is SToken, CErc20Interface {
    /**
     * @notice Initialize the new money market
     * @param underlying_ The address of the underlying asset
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     */
    function initialize(address underlying_,
                        ComptrollerInterface comptroller,
                        InterestRateModel interestRateModel_,
                        uint initialExchangeRateMantissa_,
                        string memory name,
                        string memory symbol,
                        uint8 decimals) public {
        // CToken initialize does the bulk of the work
        super.initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_, symbol_, decimals_);

        // Set underlying and sanity check it
        underlying = underlying_;
        EIP20Interface(underlying).totalSupply();
    }

    <*** User Interface ***/
```

```
    /**
     * @notice Sender supplies assets into the market and receives cTokens in exchange
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param mintAmount The amount of the underlying asset to supply
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function mint(uint mintAmount) external returns (uint) {
        (uint err,) = mintInternal(mintAmount);
        return err;
    }

    /**
     * @notice Sender redeems cTokens in exchange for the underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     * @param redeemTokens The number of cTokens to redeem into underlying
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function redeem(uint redeemTokens) external returns (uint) {
        return redeemInternal(redeemTokens);
    }

    /**
     * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset

```

```

    * @dev Accrues interest whether or not the operation succeeds, unless reverted
    * @param redeemAmount The amount of underlying to redeem
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function redeemUnderlying(uint redeemAmount) external returns (uint) {
    return redeemUnderlyingInternal(redeemAmount);
}

/**
    * @notice Sender borrows assets from the protocol to their own address
    * @param borrowAmount The amount of the underlying asset to borrow
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function borrow(uint borrowAmount) external returns (uint) {
    return borrowInternal(borrowAmount);
}

/**
    * @notice Sender repays their own borrow
    * @param repayAmount The amount to repay
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function repayBorrow(uint repayAmount) external returns (uint) {
    (uint err,) = repayBorrowInternal(repayAmount);
    return err;
}

/**
    * @notice Sender repays a borrow belonging to borrower
    * @param borrower the account with the debt being payed off
    * @param repayAmount The amount to repay
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint) {
    (uint err,) = repayBorrowBehalfInternal(borrower, repayAmount);
    return err;
}

/**
    * @notice The sender liquidates the borrowers collateral.
    * The collateral seized is transferred to the liquidator.
    * @param borrower The borrower of this cToken to be liquidated
    * @param repayAmount The amount of the underlying borrowed asset to repay
    * @param cTokenCollateral The market in which to seize collateral from the borrower
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) external
returns (uint) {
    (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
    return err;
}

/**
    * @notice The sender adds to reserves.
    * @param addAmount The amount fo underlying token to add as reserves
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function _addReserves(uint addAmount) external returns (uint) {
    return _addReservesInternal(addAmount);
}

/* *** Safe Token *** */

/**
    * @notice Gets balance of this contract in terms of the underlying
    * @dev This excludes the value of the current message, if any
    * @return The quantity of underlying tokens owned by this contract
    */
function getCashPrior() internal view returns (uint) {
    EIP20Interface token = EIP20Interface(underlying);
    return token.balanceOf(address(this));
}

/**
    * @dev Similar to EIP20 transfer, except it handles a False result from `transferFrom` and reverts in that
    * case.
    * This will revert due to insufficient balance or insufficient allowance.
    * This function returns the actual amount received,
    * which may be less than `amount` if there is a fee attached to the transfer.
    * Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value.
    * See here: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
    */
function doTransferIn(address from, uint amount) internal returns (uint) {
    EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
    uint balanceBefore = EIP20Interface(underlying).balanceOf(address(this));
}

```

```

token.transferFrom(from, address(this), amount);

bool success;
assembly {
    switch returndatasize()
        case 0 {
            success := not(0) // This is a non-standard ERC-20
        }
        case 32 {
            returndatacopy(0, 0, 32) // This is a compliant ERC-20
            success := mload(0) // Set `success = returndata` of external call
        }
        default {
            revert(0, 0) // This is an excessively non-compliant ERC-20, revert.
        }
    }
    require(success, "TOKEN_TRANSFER_IN_FAILED");

    // Calculate the amount that was *actually* transferred
    uint balanceAfter = EIP20Interface(underlying).balanceOf(address(this));
    require(balanceAfter >= balanceBefore, "TOKEN_TRANSFER_IN_OVERFLOW");
    return balanceAfter - balanceBefore; // underflow already checked above, just subtract
}

/**
 * @dev Similar to EIP20 transfer, except it handles a False success from `transfer` and returns an
 * explanatory error code rather than reverting. If caller has not called checked protocol's balance, this may revert
 * due to insufficient cash held in this contract. If caller has checked protocol's balance prior to this call, and
 * verified it is >= amount, this should not revert in normal conditions.
 *
 * Note: This wrapper safely handles non-standard ERC-20 tokens that do not return a value. See here:
 * https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
 */
function doTransferOut(address payable to, uint amount) internal {
    EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
    token.transfer(to, amount);

    bool success;
    assembly {
        switch returndatasize()
            case 0 {
                success := not(0) // This is a non-standard ERC-20
            }
            case 32 {
                returndatacopy(0, 0, 32) // This is a compliant ERC-20
                success := mload(0) // Set `success = returndata` of external call
            }
            default {
                revert(0, 0) // This is an excessively non-compliant ERC-20, revert.
            }
        }
        require(success, "TOKEN_TRANSFER_OUT_FAILED");
    }
}

CErc20Delegate.sol
pragma solidity ^0.5.16;

import "./CErc20.sol";

/**
 * @title Compound's CErc20Delegate Contract
 * @notice CTokens which wrap an EIP-20 underlying and are delegated to
 * @author Compound
 */
contract CErc20Delegate is CErc20, CDelegateInterface {
    /**
     * @notice Construct an empty delegate
     */
    constructor() public {}

    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public {
        // Shh -- currently unused
        data;
    }

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

```

```

        implementation = address(0);
    }

    require(msg.sender == admin, "only the admin may call _becomeImplementation");
}

<**
 * @notice Called by the delegator on a delegate to forfeit its responsibility
 */
function resignImplementation() public {
    // Shh -- we don't ever want this hook to be marked pure
    if(false) {
        implementation = address(0);
    }

    require(msg.sender == admin, "only the admin may call _resignImplementation");
}
}

CErc20Delegator.sol
pragma solidity ^0.5.16;

import "./CTokenInterfaces.sol";

<**
 * @title Compound's CErc20Delegator Contract
 * @notice CTokens which wrap an EIP-20 underlying and delegate to an implementation
 * @author Compound
 */
contract CErc20Delegator is CTokenInterface, CErc20Interface, CDelegatorInterface {
<**
 * @notice Construct a new money market
 * @param underlying_ The address of the underlying asset
 * @param comptroller_ The address of the Comptroller
 * @param interestRateModel_ The address of the interest rate model
 * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
 * @param name_ ERC-20 name of this token
 * @param symbol_ ERC-20 symbol of this token
 * @param decimals_ ERC-20 decimal precision of this token
 * @param admin_ Address of the administrator of this token
 * @param implementation_ The address of the implementation the contract delegates to
 * @param becomeImplementationData_ The encoded args for becomeImplementation
 */
constructor(address underlying_,
           ComptrollerInterface comptroller_,
           InterestRateModel interestRateModel_,
           uint initialExchangeRateMantissa_,
           string memory name_,
           string memory symbol_,
           uint8 decimals_,
           address payable admin_,
           address implementation_,
           bytes memory becomeImplementationData) public {
    // Creator of the contract is admin during initialization
    admin = msg.sender;

    // First delegate gets to initialize the delegator (i.e. storage contract)
    delegateTo(implementation_);
    abi.encodeWithSignature("initialize(address,address,address,uint256,string,string,uint8)",
                           underlying_,
                           comptroller_,
                           interestRateModel_,
                           initialExchangeRateMantissa_,
                           name_,
                           symbol_,
                           decimals_);
}

// New implementations always get set via the settor (post-initialize)
_setImplementation(implementation_, false, becomeImplementationData);

// Set the proper admin now that initialization is done
admin = admin_;
}

<**
 * @notice Called by the admin to update the implementation of the delegator
 * @param implementation_ The address of the new implementation for delegation
 * @param allowResign_ Flag to indicate whether to call _resignImplementation on the old implementation
 * @param becomeImplementationData_ The encoded bytes data to be passed to _becomeImplementation
 */
function setImplementation(address implementation_, bool allowResign_, bytes memory becomeImplementationData) public {
    require(msg.sender == admin, "CErc20Delegator::setImplementation: Caller must be admin");

    if(allowResign) {
}
}
}

```

```
        delegateToImplementation(abi.encodeWithSignature("_resignImplementation()"));
    }

    address oldImplementation = implementation;
    implementation = implementation_;

    delegateToImplementation(abi.encodeWithSignature("_becomeImplementation(bytes)",
becomeImplementationData));

    emit NewImplementation(oldImplementation, implementation);
}

/***
 * @notice Sender supplies assets into the market and receives cTokens in exchange
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param mintAmount The amount of the underlying asset to supply
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function mint(uint mintAmount) external returns (uint) {
    mintAmount; // Shh
    delegateAndReturn();
}

/***
 * @notice Sender redeems cTokens in exchange for the underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemTokens The number of cTokens to redeem into underlying
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function redeem(uint redeemTokens) external returns (uint) {
    redeemTokens; // Shh
    delegateAndReturn();
}

/***
 * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemAmount The amount of underlying to redeem
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function redeemUnderlying(uint redeemAmount) external returns (uint) {
    redeemAmount; // Shh
    delegateAndReturn();
}

/***
 * @notice Sender borrows assets from the protocol to their own address
 * @param borrowAmount The amount of the underlying asset to borrow
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function borrow(uint borrowAmount) external returns (uint) {
    borrowAmount; // Shh
    delegateAndReturn();
}

/***
 * @notice Sender repays their own borrow
 * @param repayAmount The amount to repay
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function repayBorrow(uint repayAmount) external returns (uint) {
    repayAmount; // Shh
    delegateAndReturn();
}

/***
 * @notice Sender repays a borrow belonging to borrower
 * @param borrower the account with the debt being payed off
 * @param repayAmount The amount to repay
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint) {
    borrower; repayAmount; // Shh
    delegateAndReturn();
}

/***
 * @notice The sender liquidates the borrowers collateral.
 * @param borrower The borrower of this cToken to be liquidated
 * @param cTokenCollateral The market in which to seize collateral from the borrower
 * @param repayAmount The amount of the underlying borrowed asset to repay
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) external
returns (uint) {
    borrower; repayAmount; cTokenCollateral; // Shh
}
```

```
delegateAndReturn();  
}  
  
/**  
 * @notice Transfer `amount` tokens from `msg.sender` to `dst`  
 * @param dst The address of the destination account  
 * @param amount The number of tokens to transfer  
 * @return Whether or not the transfer succeeded  
 */  
function transfer(address dst, uint amount) external returns (bool) {  
    dst; amount; // Shh  
    delegateAndReturn();  
}  
  
/**  
 * @notice Transfer `amount` tokens from `src` to `dst`  
 * @param src The address of the source account  
 * @param dst The address of the destination account  
 * @param amount The number of tokens to transfer  
 * @return Whether or not the transfer succeeded  
 */  
function transferFrom(address src, address dst, uint256 amount) external returns (bool) {  
    src; dst; amount; // Shh  
    delegateAndReturn();  
}  
  
/**  
 * @notice Approve `spender` to transfer up to `amount` from `src`  
 * @dev This will overwrite the approval amount for `spender`  
 * and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)  
 * @param spender The address of the account which may transfer tokens  
 * @param amount The number of tokens that are approved (-1 means infinite)  
 * @return Whether or not the approval succeeded  
 */  
function approve(address spender, uint256 amount) external returns (bool) {  
    spender; amount; // Shh  
    delegateAndReturn();  
}  
  
/**  
 * @notice Get the current allowance from `owner` for `spender`  
 * @param owner The address of the account which owns the tokens to be spent  
 * @param spender The address of the account which may transfer tokens  
 * @return The number of tokens allowed to be spent (-1 means infinite)  
 */  
function allowance(address owner, address spender) external view returns (uint) {  
    owner; spender; // Shh  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Get the token balance of the `owner`  
 * @param owner The address of the account to query  
 * @return The number of tokens owned by `owner`  
 */  
function balanceOf(address owner) external view returns (uint) {  
    owner; // Shh  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Get the underlying balance of the `owner`  
 * @dev This also accrues interest in a transaction  
 * @param owner The address of the account to query  
 * @return The amount of underlying owned by `owner`  
 */  
function balanceOfUnderlying(address owner) external returns (uint) {  
    owner; // Shh  
    delegateAndReturn();  
}  
  
/**  
 * @notice Get a snapshot of the account's balances, and the cached exchange rate  
 * @dev This is used by comptroller to more efficiently perform liquidity checks.  
 * @param account Address of the account to snapshot  
 * @return (possible error, token balance, borrow balance, exchange rate mantissa)  
 */  
function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint) {  
    account; // Shh  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Returns the current per-block borrow interest rate for this cToken  
 * @return The borrow interest rate per block, scaled by 1e18  
 */  
function borrowRatePerBlock() external view returns (uint) {
```

```
delegateToViewAndReturn();  
}  
  
/**  
 * @notice Returns the current per-block supply interest rate for this cToken  
 * @return The supply interest rate per block, scaled by 1e18  
 */  
function supplyRatePerBlock() external view returns (uint) {  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Returns the current total borrows plus accrued interest  
 * @return The total borrows with interest  
 */  
function totalBorrowsCurrent() external returns (uint) {  
    delegateAndReturn();  
}  
  
/**  
 * @notice Accrue interest to updated borrowIndex and then calculate account's borrow balance using the  
 * updated borrowIndex  
 * @param account The address whose balance should be calculated after updating borrowIndex  
 * @return The calculated balance  
 */  
function borrowBalanceCurrent(address account) external returns (uint) {  
    account; // Shh  
    delegateAndReturn();  
}  
  
/**  
 * @notice Return the borrow balance of account based on stored data  
 * @param account The address whose balance should be calculated  
 * @return The calculated balance  
 */  
function borrowBalanceStored(address account) public view returns (uint) {  
    account; // Shh  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Accrue interest then return the up-to-date exchange rate  
 * @return Calculated exchange rate scaled by 1e18  
 */  
function exchangeRateCurrent() public returns (uint) {  
    delegateAndReturn();  
}  
  
/**  
 * @notice Calculates the exchange rate from the underlying to the CToken  
 * @dev This function does not accrue interest before calculating the exchange rate  
 * @return Calculated exchange rate scaled by 1e18  
 */  
function exchangeRateStored() public view returns (uint) {  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Get cash balance of this cToken in the underlying asset  
 * @return The quantity of underlying asset owned by this contract  
 */  
function getCash() external view returns (uint) {  
    delegateToViewAndReturn();  
}  
  
/**  
 * @notice Applies accrued interest to total borrows and reserves.  
 * @dev This calculates interest accrued from the last checkpointed block  
 *      up to the current block and writes new checkpoint to storage.  
 */  
function accrueInterest() public returns (uint) {  
    delegateAndReturn();  
}  
  
/**  
 * @notice Transfers collateral tokens (this market) to the liquidator.  
 * @dev Will fail unless called by another cToken during the process of liquidation.  
 * Its absolutely critical to use msg.sender as the borrowed cToken and not a parameter.  
 * @param liquidator The account receiving seized collateral  
 * @param borrower The account having collateral seized  
 * @param seizeTokens The number of cTokens to seize  
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)  
 */  
function seize(address liquidator, address borrower, uint seizeTokens) external returns (uint) {  
    liquidator; borrower; seizeTokens; // Shh  
    delegateAndReturn();  
}
```

```
/** Admin Functions */
/*
 * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
 * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to finalize the transfer.
 * @param newPendingAdmin New pending admin.
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _setPendingAdmin(address payable newPendingAdmin) external returns (uint) {
    newPendingAdmin; // Shh
    delegateAndReturn();
}

/*
 * @notice Sets a new comptroller for the market
 * @dev Admin function to set a new comptroller
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _setComptroller(ComptrollerInterface newComptroller) public returns (uint) {
    newComptroller; // Shh
    delegateAndReturn();
}

/*
 * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
 * @dev Admin function to accrue interest and set a new reserve factor
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _setReserveFactor(uint newReserveFactorMantissa) external returns (uint) {
    newReserveFactorMantissa; // Shh
    delegateAndReturn();
}

/*
 * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
 * @dev Admin function for pending admin to accept role and update admin
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _acceptAdmin() external returns (uint) {
    delegateAndReturn();
}

/*
 * @notice Accrues interest and adds reserves by transferring from admin
 * @param addAmount Amount of reserves to add
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _addReserves(uint addAmount) external returns (uint) {
    addAmount; // Shh
    delegateAndReturn();
}

/*
 * @notice Accrues interest and reduces reserves by transferring to admin
 * @param reduceAmount Amount of reduction to reserves
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _reduceReserves(uint reduceAmount) external returns (uint) {
    reduceAmount; // Shh
    delegateAndReturn();
}

/*
 * @notice Accrues interest and updates the interest rate model using _setInterestRateModelFresh
 * @dev Admin function to accrue interest and update the interest rate model
 * @param newInterestRateModel the new interest rate model to use
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
    newInterestRateModel; // Shh
    delegateAndReturn();
}

/*
 * @notice Internal method to delegate execution to another contract
 * @dev It returns to the external caller whatever the implementation returns or forwards reverts
 * @param callee The contract to delegatecall
 * @param data The raw data to delegatecall
 * @return The returned bytes from the delegatecall
*/
function delegateTo(address callee, bytes memory data) internal returns (bytes memory) {
    (bool success, bytes memory returnData) = callee.delegatecall(data);
    assembly {
        if eq(success, 0) {

```

```

        revert(add(returnData, 0x20), returndatasize)
    }
    return returnData;
}

/**
 * @notice Delegates execution to the implementation contract
 * @dev It returns to the external caller whatever the implementation returns or forwards reverts
 * @param data The raw data to delegatecall
 * @return The returned bytes from the delegatecall
 */
function delegateToImplementation(bytes memory data) public returns (bytes memory) {
    return delegateTo(implementation, data);
}

/**
 * @notice Delegates execution to an implementation contract
 * @dev It returns to the external caller whatever the implementation returns or forwards reverts
 * There are an additional 2 prefix uints from the wrapper returndata, which we ignore since we make an
extra hop
 * @param data The raw data to delegatecall
 * @return The returned bytes from the delegatecall
*/
function delegateToViewImplementation(bytes memory data) public view returns (bytes memory) {
    (bool success, bytes memory returnData) =
address(this).staticcall(abi.encodeWithSignature("delegateToImplementation(bytes)", data));
    assembly {
        if eq(success, 0) {
            revert(add(returnData, 0x20), returndatasize)
        }
    }
    return abi.decode(returnData, (bytes));
}

function delegateToViewAndReturn() private view returns (bytes memory) {
    (bool success, ) = address(this).staticcall(abi.encodeWithSignature("delegateToImplementation(bytes)", msg.data));
    assembly {
        let free_mem_ptr := mload(0x40)
        returndatacopy(free_mem_ptr, 0, returndatasize)

        switch success
        case 0 { revert(free_mem_ptr, returndatasize) }
        default { return(add(free_mem_ptr, 0x40), returndatasize) }
    }
}

function delegateAndReturn() private returns (bytes memory) {
    (bool success, ) = implementation.delegatecall(msg.data);
    assembly {
        let free_mem_ptr := mload(0x40)
        returndatacopy(free_mem_ptr, 0, returndatasize)

        switch success
        case 0 { revert(free_mem_ptr, returndatasize) }
        default { return(free_mem_ptr, returndatasize) }
    }
}

/**
 * @notice Delegates execution to an implementation contract
 * @dev It returns to the external caller whatever the implementation returns or forwards reverts
 */
function () external payable {
    require(msg.value == 0, "CErc20Delegator:fallback: cannot send value to fallback");

    // delegate all other functions to current implementation
    delegateAndReturn();
}

```

*CErc20Immutable.sol*

```

pragma solidity ^0.5.16;
import "./CErc20.sol";

/**
 * @title Compound's CErc20Immutable Contract
 * @notice CTokens which wrap an EIP-20 underlying and are immutable
 * @author Compound
 */
contract CErc20Immutable is CErc20 {

```

```


/**
 * @notice Construct a new money market
 * @param underlying_ The address of the underlying asset
 * @param comptroller_ The address of the Comptroller
 * @param interestRateModel_ The address of the interest rate model
 * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
 * @param name_ ERC-20 name of this token
 * @param symbol_ ERC-20 symbol of this token
 * @param decimals_ ERC-20 decimal precision of this token
 * @param admin_ Address of the administrator of this token
 */
constructor(address underlying_,
            ComptrollerInterface comptroller_,
            InterestRateModel interestRateModel_,
            uint initialExchangeRateMantissa_,
            string memory name_,
            string memory symbol_,
            uint8 decimals_,
            address payable admin_) public {
    // Creator of the contract is admin during initialization
    admin = msg.sender;

    // Initialize the market
    initialize(underlying_, comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_,
              symbol_, decimals_);

    // Set the proper admin now that initialization is done
    admin = admin_;
}
}

CEther.sol
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./SToken.sol";

/**
 * @title Compound's CEther Contract
 * @notice CToken which wraps Ether
 * @author Compound
 */
contract CEther is SToken {
    /**
     * @notice Construct a new CEther money market
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     * @param admin_ Address of the administrator of this token
     */
    constructor(ComptrollerInterface comptroller_,
               InterestRateModel interestRateModel_,
               uint initialExchangeRateMantissa_,
               string memory name_,
               string memory symbol_,
               uint8 decimals_,
               address payable admin_) public {
        // Creator of the contract is admin during initialization
        admin = msg.sender;

        initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_, name_, symbol_,
                  decimals_);

        // Set the proper admin now that initialization is done
        admin = admin_;
    }

    /**
     * @notice Sender supplies assets into the market and receives cTokens in exchange
     * @dev Reverts upon any failure
     */
    function mint() external payable {
        (uint err,) = mintInternal(msg.value);
        requireNoError(err, "mint failed");
    }

    /**
     * @notice Sender redeems cTokens in exchange for the underlying asset
     * @dev Accrues interest whether or not the operation succeeds, unless reverted
     */
}


```

```

    * @param redeemTokens The number of cTokens to redeem into underlying
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function redeem(uint redeemTokens) external returns (uint) {
    return redeemInternal(redeemTokens);
}

/**
    * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
    * @dev Accrues interest whether or not the operation succeeds, unless reverted
    * @param redeemAmount The amount of underlying to redeem
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function redeemUnderlying(uint redeemAmount) external returns (uint) {
    return redeemUnderlyingInternal(redeemAmount);
}

/**
    * @notice Sender borrows assets from the protocol to their own address
    * @param borrowAmount The amount of the underlying asset to borrow
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
function borrow(uint borrowAmount) external returns (uint) {
    return borrowInternal(borrowAmount);
}

/**
    * @notice Sender repays their own borrow
    * @dev Reverts upon any failure
    */
function repayBorrow() external payable {
    (uint err,) = repayBorrowInternal(msg.value);
    requireNoError(err, "repayBorrow failed");
}

/**
    * @notice Sender repays a borrow belonging to borrower
    * @dev Reverts upon any failure
    * @param borrower the account with the debt being payed off
    */
function repayBorrowBehalf(address borrower) external payable {
    (uint err,) = repayBorrowBehalfInternal(borrower, msg.value);
    requireNoError(err, "repayBorrowBehalf failed");
}

/**
    * @notice The sender liquidates the borrowers collateral.
    * The collateral seized is transferred to the liquidator.
    * @dev Reverts upon any failure
    * @param borrower The borrower of this cToken to be liquidated
    * @param cTokenCollateral The market in which to seize collateral from the borrower
    */
function liquidateBorrow(address borrower, CToken cTokenCollateral) external payable {
    (uint err,) = liquidateBorrowInternal(borrower, msg.value, cTokenCollateral);
    requireNoError(err, "liquidateBorrow failed");
}

/**
    * @notice Send Ether to CEther to mint
    */
function () external payable {
    (uint err,) = mintInternal(msg.value);
    requireNoError(err, "mint failed");
}

/* *** Safe Token *** */

/**
    * @notice Gets balance of this contract in terms of Ether, before this message
    * @dev This excludes the value of the current message, if any
    * @return The quantity of Ether owned by this contract
    */
function getCashPrior() internal view returns (uint) {
    (MathError err, uint startingBalance) = subUInt(address(this).balance, msg.value);
    require(err == MathError.NO_ERROR);
    return startingBalance;
}

/**
    * @notice Perform the actual transfer in, which is a no-op
    * @param from Address sending the Ether
    * @param amount Amount of Ether being sent
    * @return The actual amount of Ether transferred
    */
function doTransferIn(address from, uint amount) internal returns (uint) {
    // Sanity checks
    require(msg.sender == from, "sender mismatch");
}

```

```

require(msg.value == amount, "value mismatch");
return amount;
}

function doTransferOut(address payable to, uint amount) internal {
    /* Send the Ether, with minimal gas and revert on failure */
    to.transfer(amount);
}

function requireNoError(uint errCode, string memory message) internal pure {
    if (errCode == uint(Error.NO_ERROR)) {
        return;
    }

    bytes memory fullMessage = new bytes(bytes(message).length + 5);
    uint i;

    for (i = 0; i < bytes(message).length; i++) {
        fullMessage[i] = bytes(message)[i];
    }

    fullMessage[i+0] = byte(uint8(32));
    fullMessage[i+1] = byte(uint8(40));
    fullMessage[i+2] = byte(uint8(48 + (errCode / 10)));
    fullMessage[i+3] = byte(uint8(48 + (errCode % 10)));
    fullMessage[i+4] = byte(uint8(41));

    require(errCode == uint(Error.NO_ERROR), string(fullMessage));
}

function isNativeToken() public pure returns (bool) {
    return true;
}
}

```

### Comptroller.sol

```

pragma solidity ^0.5.16;

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


$$\begin{array}{c} \text{* * *} \\ \text{* @title Compound's Comptroller Contract} \\ \text{* @author Compound} \\ \text{* */} \end{array}$$


contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    ///@notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

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

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

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

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

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

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

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

    ///@notice Emitted when pause guardian is changed
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

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

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

```

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

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

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

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

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

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

/// @notice The initial COMP index for a market
uint24 public constant compInitialIndex = 1e36;

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

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

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

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

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

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

/** Assets You Are In **/ 

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

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

/**
 * @notice Add assets to be included in account liquidity calculation
 * @param cTokens The list of addresses of the cToken markets to be enabled
 * @return Success indicator for whether each corresponding market was entered
 */
function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
    uint len = cTokens.length;
    uint[] memory results = new uint[](len);
    for (uint i = 0; i < len; i++) {
        CToken cToken = CToken(cTokens[i]);
        results[i] = uint(addToMarketInternal(cToken, msg.sender));
    }
    return results;
}

/**
 * @notice Add the market to the borrower's "assets in" for liquidity calculations
 * @param cToken The market to enter

```

```

    * @param borrower The address of the account to modify
    * @return Success indicator for whether the market was entered
    */
function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
    Market storage marketToJoin = markets[address(cToken)];
    if (!marketToJoin.isListed) {
        // market is not listed, cannot join
        return Error.MARKET_NOT_LISTED;
    }
    if (marketToJoin.accountMembership[borrower] == true) {
        // already joined
        return Error.NO_ERROR;
    }
    if (accountAssets[borrower].length >= maxAssets) {
        // no space, cannot join
        return Error.TOO_MANY_ASSETS;
    }
    // survived the gauntlet, add to list
    // NOTE: we store these somewhat redundantly as a significant optimization
    // this avoids having to iterate through the list for the most common use cases
    // that is, only when we need to perform liquidity checks
    // and not whenever we want to check if an account is in a particular market
    marketToJoin.accountMembership[borrower] = true;
    accountAssets[borrower].push(cToken);
    emit MarketEntered(cToken, borrower);
    return Error.NO_ERROR;
}

/**
 * @notice Removes asset from sender's account liquidity calculation
 * @dev Sender must not have an outstanding borrow balance in the asset,
 * or be providing necessary collateral for an outstanding borrow.
 * @param cTokenAddress The address of the asset to be removed
 * @return Whether or not the account successfully exited the market
 */
function exitMarket(address cTokenAddress) external returns (uint) {
    CToken cToken = CToken(cTokenAddress);
    /* Get sender tokensHeld and amountOwed underlying from the cToken */
    (uint oErr, uint tokensHeld, uint amountOwed,) = cToken.getAccountSnapshot(msg.sender);
    require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code
    /* Fail if the sender has a borrow balance */
    if (amountOwed != 0) {
        return fail(Error.NONZERO_BORROW_BALANCE,
FailureInfo.EXIT_MARKET_BALANCE_OWED);
    }
    /* Fail if the sender is not permitted to redeem all of their tokens */
    uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
    if (allowed != 0) {
        return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
    }
    Market storage marketToExit = markets[address(cToken)];
    /* Return true if the sender is not already 'in' the market */
    if (!marketToExit.accountMembership[msg.sender]) {
        return uint(Error.NO_ERROR);
    }
    /* Set cToken account membership to false */
    delete marketToExit.accountMembership[msg.sender];
    /* Delete cToken from the account's list of assets */
    // load into memory for faster iteration
    CToken[] memory userAssetList = accountAssets[msg.sender];
    uint len = userAssetList.length;
    uint assetIndex = len;
    for (uint i = 0; i < len; i++) {
        if (userAssetList[i] == cToken) {
            assetIndex = i;
            break;
        }
    }
    // We *must* have found the asset in the list or our redundant data structure is broken
    assert(assetIndex < len);
    // copy last item in list to location of item to be removed, reduce length by 1
    CToken[] storage storedList = accountAssets[msg.sender];
    storedList[assetIndex] = storedList[storedList.length - 1];
}

```

```

        storedList.length--;
        emit MarketExited(cToken, msg.sender);
        return uint(Error.NO_ERROR);
    }

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

        // Shh - currently unused
        minter;
        mintAmount;

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

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

        return uint(Error.NO_ERROR);
    }

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

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

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

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

        return uint(Error.NO_ERROR);
    }

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

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

        /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
    }
}

```

```

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

/**
 * @notice Validates redeem and reverts on rejection. May emit logs.
 * @param cToken Asset being redeemed
 * @param redeemer The address redeeming the tokens
 * @param redeemAmount The amount of the underlying asset being redeemed
 * @param redeemTokens The number of tokens being redeemed
 */
function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
    // Shh - currently unused
    cToken;
    redeemer;

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

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

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

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

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

        // it should be impossible to break the important invariant
        assert(markets[cToken].accountMembership[borrower]);
    }

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

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

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

    return uint(Error.NO_ERROR);
}

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

```

```
// Shh - currently unused
cToken;
borrower;
borrowAmount;

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

/**
* @notice Checks if the account should be allowed to repay a borrow in the given market
* @param cToken The market to verify the repay against
* @param payer The account which would repay the asset
* @param borrower The account which would borrowed the asset
* @param repayAmount The amount of the underlying asset the account would repay
* @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
*/
function repayBorrowAllowed(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount) external returns (uint) {
    // Shh - currently unused
    payer;
    borrower;
    repayAmount;

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

```

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

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

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

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

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

    // Shh - currently unused
    seizeTokens;

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

    if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
        return uint(Error.COMPTROLLER_MISMATCH);
    }

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

    return uint(Error.NO_ERROR);
}

```

```

}

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

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

/***
 * @notice Checks if the account should be allowed to transfer tokens in the given market
 * @param cToken The market to verify the transfer against
 * @param src The account which sources the tokens
 * @param dst The account which receives the tokens
 * @param transferTokens The number of cTokens to transfer
 * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint)
{
    // Pausing is a very serious situation - we revert to sound the alarms
    require(!transferGuardianPaused, "transfer is paused");

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

/***
 * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
 * Note that `cTokenBalance` is the number of cTokens the account owns in the market,
 * whereas `borrowBalance` is the amount of underlying that the account has borrowed.
 */
struct AccountLiquidityLocalVars {
    uint sumCollateral;
    uint sumBorrowPlusEffects;
}

```

```

        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

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

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

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
     *         hypothetical account liquidity in excess of collateral requirements,
     *         hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account,
CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

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

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

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

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account);
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
upgrades
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
        }
    }
}

```

```

vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

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

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

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

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

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

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

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

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

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

```

```

Exp memory numerator;
Exp memory denominator;
Exp memory ratio;
MathError mathErr;

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

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

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

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

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

/** Admin Functions **/


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

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sets the collateralFactor for a market
 * @dev Admin function to set per-market collateralFactor
 * @param cToken The market to set the factor on
 */

```

```

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

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

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

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

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

```

```

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

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

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

return uint(Error.NO_ERROR);
}

/**
 * @notice Add the market to the markets mapping and set it as listed
 * @dev Admin function to set isListed and add support for the market
 * @param cToken The address of the market (token) to list
 * @return uint 0=success, otherwise a failure. (See enum Error for details)
 */
function _supportMarket(CToken cToken) external returns (uint) {
    if(msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
    }

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

    cToken.isCToken(); // Sanity check to make sure its really a CToken
    markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});
    _addMarketInternal(address(cToken));
    emit MarketListed(cToken);
    return uint(Error.NO_ERROR);
}

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

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

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

    // Store pauseGuardian with value newPauseGuardian
    pauseGuardian = newPauseGuardian;

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

```

```

}

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

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

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

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

/** Comp Distribution **/

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

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

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

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

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

```

```

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

/**
 * @notice Accrue COMP to the market by updating the borrow index
 * @param cToken The market whose borrow index to update
 */
function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
    CompMarketState storage borrowState = compBorrowState[cToken];
    uint borrowSpeed = compSpeeds[cToken];
    uint blockNumber = getBlockNumber();
    uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
    if (deltaBlocks > 0 && borrowSpeed > 0) {
        uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
        uint compAccrued = mul_(deltaBlocks, borrowSpeed);
        Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) :
Double({mantissa: 0});
        Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
        compBorrowState[cToken] = CompMarketState({
            index: safe224(index.mantissa, "new index exceeds 224 bits"),
            block: safe32(blockNumber, "block number exceeds 32 bits")
        });
    } else if (deltaBlocks > 0) {
        borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
    }
}

/**
 * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
 * @param cToken The market in which the supplier is interacting
 * @param supplier The address of the supplier to distribute COMP to
 */
function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
    CompMarketState storage supplyState = compSupplyState[cToken];
    Double memory supplyIndex = Double({mantissa: supplyState.index});
    Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][supplier]});
    compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;
    if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
        supplierIndex.mantissa = compInitialIndex;
    }

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

/**
 * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
 * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
 * @param cToken The market in which the borrower is interacting
 * @param borrower The address of the borrower to distribute COMP to
 */
function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex, bool
distributeAll) internal {
    CompMarketState storage borrowState = compBorrowState[cToken];
    Double memory borrowIndex = Double({mantissa: borrowState.index});
    Double memory borrowerIndex = Double({mantissa: compBorrowerIndex[cToken][borrower]});
    compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

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

function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint);

```

```

/*
 * @notice Transfer COMP to the user, if they are above the threshold
 * @dev Note: If there is not enough COMP, we do not perform the transfer at all.
 * @param user The address of the user to transfer COMP to
 * @param userAccrued The amount of COMP to (possibly) transfer
 * @return The amount of COMP which was NOT transferred to the user
 */

function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
    if(userAccrued >= threshold && userAccrued > 0) {
        Comp comp = Comp(getCompAddress());
        uint compRemaining = comp.balanceOf(address(this));
        if(userAccrued <= compRemaining) {
            comp.transfer(user, userAccrued);
        }
    }
    return userAccrued;
}

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

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

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

*** Comp Distribution Admin ***

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

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

    refreshCompSpeedsInternal();
}

/*
 * @notice Add markets to compMarkets, allowing them to earn COMP in the flywheel
 * @param cTokens The addresses of the markets to add
 */
function _addCompMarkets(address[] memory cTokens) public {


```

```

require(adminOrInitializing(), "only admin can add comp market");

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

refreshCompSpeedsInternal();
}

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

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

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

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

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

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

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

    refreshCompSpeedsInternal();
}

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

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

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

```

### ComptrollerG1.sol

```

pragma solidity ^0.5.16;

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

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 * @dev This was the first version of the Comptroller brains.

```

```
* We keep it so our tests can continue to do the real-life behavior of upgrading from this logic forward.
*/
contract ComptrollerG1 is ComptrollerV1Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    struct Market {
        /**
         * @notice Whether or not this market is listed
         */
        bool isListed;

        /**
         * @notice Multiplier representing the most one can borrow against their collateral in this market.
         * For instance, 0.9 to allow borrowing 90% of collateral value.
         * Must be between 0 and 1, and stored as a mantissa.
         */
        uint collateralFactorMantissa;

        /**
         * @notice Per-market mapping of "accounts in this asset"
         */
        mapping(address => bool) accountMembership;
    }

    /**
     * @notice Official mapping of cTokens -> Market metadata
     * @dev Used e.g. to determine if a market is supported
     */
    mapping(address => Market) public markets;

    /**
     * @notice Emitted when an admin supports a market
     */
    event MarketListed(CToken cToken);

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

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

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

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

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

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

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

    // closeFactorMantissa must be strictly greater than this value
    uint constant closeFactorMinMantissa = 5e16; // 0.05

    // closeFactorMantissa must not exceed this value
    uint constant closeFactorMaxMantissa = 9e17; // 0.9

    // liquidationIncentiveMantissa must be no less than this value
    uint constant liquidationIncentiveMinMantissa = mantissaOne;

    // liquidationIncentiveMantissa must be no greater than this value
    uint constant liquidationIncentiveMaxMantissa = 15e17; // 1.5

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

```

}

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

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

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

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

        if (!marketToJoin.isListed) {
            // if market is not listed, cannot join move along
            results[i] = uint(Error.MARKET_NOT_LISTED);
            continue;
        }

        if (marketToJoin.accountMembership[msg.sender] == true) {
            // if already joined, move along
            results[i] = uint(Error.NO_ERROR);
            continue;
        }

        if (accountAssets[msg.sender].length >= maxAssets) {
            // if no space, cannot join, move along
            results[i] = uint(Error.TOO_MANY_ASSETS);
            continue;
        }

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

        emit MarketEntered(cToken, msg.sender);
        results[i] = uint(Error.NO_ERROR);
    }

    return results;
}

/** 
 * @notice Removes asset from sender's account liquidity calculation
 * @dev Sender must not have an outstanding borrow balance in the asset,
 * or be providing necessary collateral for an outstanding borrow.
 * @param cTokenAddress The address of the asset to be removed
 * @return Whether or not the account successfully exited the market
 */
function exitMarket(address cTokenAddress) external returns (uint) {
    CToken cToken = CToken(cTokenAddress);
    /* Get sender tokensHeld and amountOwed underlying from the cToken */
    (uint oErr, uint tokensHeld, uint amountOwed,) = cToken.getAccountSnapshot(msg.sender);
    require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

    /* Fail if the sender has a borrow balance */
    if (amountOwed != 0) {
}

```

```

        return
FailureInfo.EXIT_MARKET_BALANCE_OWED);
    }

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

    Market storage marketToExit = markets[address(cToken)];
    /* Return true if the sender is not already 'in' the market */
    if (!marketToExit.accountMembership[msg.sender]) {
        return uint(Error.NO_ERROR);
    }

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

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

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

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

    emit MarketExited(cToken, msg.sender);
    return uint(Error.NO_ERROR);
}

/***
 * @notice Checks if the account should be allowed to mint tokens in the given market
 * @param cToken The market to verify the mint against
 * @param minter The account which would get the minted tokens
 * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
 * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
    minter;           // currently unused
    mintAmount;       // currently unused

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

    // *may include Policy Hook-type checks
    return uint(Error.NO_ERROR);
}

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

    if (false) {
        maxAssets = maxAssets; // not pure
    }
}

/***
 * @notice Checks if the account should be allowed to redeem tokens in the given market
 */

```

```

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

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

    // *may include Policy Hook-type checks

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Validates redeem and reverts on rejection. May emit logs.
 * @param cToken Asset being redeemed
 * @param redeemer The address redeeming the tokens
 * @param redeemAmount The amount of the underlying asset being redeemed
 * @param redeemTokens The number of tokens being redeemed
 */
function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
    cToken;           // currently unused
    redeemer;         // currently unused
    redeemAmount;     // currently unused
    redeemTokens;     // currently unused

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

/**
 * @notice Checks if the account should be allowed to borrow the underlying asset of the given market
 * @param cToken The market to verify the borrow against
 * @param borrower The account which would borrow the asset
 * @param borrowAmount The amount of underlying the account would borrow
 * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint) {
    if (!markets[cToken].isListed) {
        return uint(Error.MARKET_NOT_LISTED);
    }

    // *may include Policy Hook-type checks

    if (!markets[cToken].accountMembership[borrower]) {
        return uint(Error.MARKET_NOT_ENTERED);
    }

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

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

    return uint(Error.NO_ERROR);
}

```

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

    if (false) {
        maxAssets = maxAssets; // not pure
    }
}

/*
 * @notice Checks if the account should be allowed to repay a borrow in the given market
 * @param cToken The market to verify the repay against
 * @param payer The account which would repay the asset
 * @param borrower The account which would borrowed the asset
 * @param repayAmount The amount of the underlying asset the account would repay
 * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function repayBorrowAllowed(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount) external returns (uint) {
    payer;           // currently unused
    borrower;        // currently unused
    repayAmount;     // currently unused

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

    // *may include Policy Hook-type checks
    return uint(Error.NO_ERROR);
}

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

    if (false) {
        maxAssets = maxAssets; // not pure
    }
}

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

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

```

        }

        // *may include Policy Hook-type checks

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

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

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount,
        uint seizeTokens) external {
        cTokenBorrowed; // currently unused
        cTokenCollateral; // currently unused
        liquidator; // currently unused
        borrower; // currently unused
        repayAmount; // currently unused
        seizeTokens; // currently unused

        if (false) {
            maxAssets = maxAssets; // not pure
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        liquidator; // currently unused
        borrower; // currently unused
        seizeTokens; // currently unused

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

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
            return uint(Error.COMPTROLLER_MISMATCH);
        }

        // *may include Policy Hook-type checks
        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     */

```

```

    * @param cTokenCollateral Asset which was used as collateral and will be seized
    * @param cTokenBorrowed Asset which was borrowed by the borrower
    * @param liquidator The address repaying the borrow and seizing the collateral
    * @param borrower The address of the borrower
    * @param seizeTokens The number of collateral tokens to seize
    */
function seizeVerify(
    address cTokenCollateral,
    address cTokenBorrowed,
    address liquidator,
    address borrower,
    uint seizeTokens) external {
    cTokenCollateral; // currently unused
    cTokenBorrowed; // currently unused
    liquidator; // currently unused
    borrower; // currently unused
    seizeTokens; // currently unused

    if (false) {
        maxAssets = maxAssets; // not pure
    }
}

/**
 * @notice Checks if the account should be allowed to transfer tokens in the given market
 * @param cToken The market to verify the transfer against
 * @param src The account which sources the tokens
 * @param dst The account which receives the tokens
 * @param transferTokens The number of cTokens to transfer
 * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint)
{
    cToken; // currently unused
    src; // currently unused
    dst; // currently unused
    transferTokens; // currently unused

    // *may include Policy Hook-type checks
    // Currently the only consideration is whether or not
    // the src is allowed to redeem this many tokens
    return redeemAllowedInternal(cToken, src, transferTokens);
}

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

    if (false) {
        maxAssets = maxAssets; // not pure
    }
}

/***
 * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
 * Note that `cTokenBalance` is the number of cTokens the account owns in the market,
 * whereas `borrowBalance` is the amount of underlying that the account has borrowed.
 */
struct AccountLiquidityLocalVars {
    uint sumCollateral;
    uint sumBorrowPlusEffects;
    uint cTokenBalance;
    uint borrowBalance;
    uint exchangeRateMantissa;
    uint oraclePriceMantissa;
    Exp collateralFactor;
    Exp exchangeRate;
    Exp oraclePrice;
    Exp tokensToEther;
}

/**
 * @notice Determine the current account liquidity wrt collateral requirements
 * @return (possible error code (semi-opaque),
 *         account liquidity in excess of collateral requirements,

```

```

/*
 *      account shortfall below collateral requirements)
 */
function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
    (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(0),
0, 0);
    return (uint(err), liquidity, shortfall);
}

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

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

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

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

        // Read the balances and exchange rate from the cToken
        (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account);
        if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
upgrades
            return (Error.SNAPSHOT_ERROR, 0, 0);
        }
        vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
        vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

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

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

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

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

        // Calculate effects of interacting with cTokenModify
        if (asset == cTokenModify) {
            // redeem effect

```

```

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

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

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

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

    /*
     * Get the exchange rate and calculate the number of collateral tokens to seize:
     * seizeAmount = repayAmount * liquidationIncentive * priceBorrowed / priceCollateral
     * seizeTokens = seizeAmount / exchangeRate
     *             = repayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
     */
    uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on
error
    uint seizeTokens;
    Exp memory numerator;
    Exp memory denominator;
    Exp memory ratio;
    MathError mathErr;

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

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

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

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

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

/** Admin Functions ***/

/**
 * @notice Sets a new price oracle for the comptroller
 * @dev Admin function to set a new price oracle
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _setPriceOracle(PriceOracle newOracle) public returns (uint) {

```

```

// Check caller is admin OR currently initializing as new unitroller implementation
if (!adminOrInitializing()) {
    return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
}

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

// Ensure invoke newOracle.isPriceOracle() returns true
// require(newOracle.isPriceOracle(), "oracle method isPriceOracle returned false");

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

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

return uint(Error.NO_ERROR);
}

/**
 * @notice Sets the closeFactor used when liquidating borrows
 * @dev Admin function to set closeFactor
 * @param newCloseFactorMantissa New close factor, scaled by 1e18
 * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
 */
function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint256) {
    // Check caller is admin OR currently initializing as new unitroller implementation
    if (!adminOrInitializing()) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
    }

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

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sets the collateralFactor for a market
 * @dev Admin function to set per-market collateralFactor
 * @param cToken The market to set the factor on
 * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
 * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
 */
function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint256) {
    // Check caller is admin
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
    }

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

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

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

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

    // Set market's collateral factor to new collateral factor, remember old value
}

```

```

    uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
    market.collateralFactorMantissa = newCollateralFactorMantissa;

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

    return uint(Error.NO_ERROR);
}

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

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

    return uint(Error.NO_ERROR);
}

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

    // Check de-scaled 1 <= newLiquidationDiscount <= 1.5
    Exp memory newLiquidationIncentive = Exp({mantissa: newLiquidationIncentiveMantissa});
    Exp memory minLiquidationIncentive = Exp({mantissa: liquidationIncentiveMinMantissa});
    if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
        return
            fail(Error.INVALID_LIQUIDATION_INCENTIVE,
            FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
    }

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

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

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Add the market to the markets mapping and set it as listed
 * @dev Admin function to set isListed and add support for the market
 * @param cToken The address of the market (token) to list
 * @return uint 0=success, otherwise a failure. (See enum Error for details)
 */
function _supportMarket(CToken cToken) external returns (uint) {
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
    }

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

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

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

```

```

        return uint(Error.NO_ERROR);
    }

    function become(Unitroller unitroller, PriceOracle _oracle, uint _closeFactorMantissa, uint _maxAssets,
bool reinitializing) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        uint changeStatus = unitroller._acceptImplementation();

        require(changeStatus == 0, "change not authorized");

        if (!reinitializing) {
            ComptrollerG1 freshBrainedComptroller = ComptrollerG1(address(unitroller));

            // Ensure invoke_setPriceOracle() = 0
            uint err = freshBrainedComptroller.setPriceOracle(_oracle);
            require (err == uint(Error.NO_ERROR), "set price oracle error");

            // Ensure invoke_setCloseFactor() = 0
            err = freshBrainedComptroller.setCloseFactor(_closeFactorMantissa);
            require (err == uint(Error.NO_ERROR), "set close factor error");

            // Ensure invoke_setMaxAssets() = 0
            err = freshBrainedComptroller.setMaxAssets(_maxAssets);
            require (err == uint(Error.NO_ERROR), "set max assets error");

            // Ensure invoke_setLiquidationIncentive(liquidationIncentiveMinMantissa) = 0
            err = freshBrainedComptroller.setLiquidationIncentive(liquidationIncentiveMinMantissa);
            require (err == uint(Error.NO_ERROR), "set liquidation incentive error");
        }
    }

    /**
     * @dev Check that caller is admin or this contract is initializing itself as
     * the new implementation.
     * There should be no way to satisfy msg.sender == comptrollerImplementation
     * without tx.origin also being admin, but both are included for extra safety
     */
    function adminOrInitializing() internal view returns (bool) {
        bool initializing = (
            msg.sender == comptrollerImplementation
            &&
            // solium-disable-next-line security/no-tx-origin
            tx.origin == admin
        );
        bool isAdmin = msg.sender == admin;
        return isAdmin || initializing;
    }
}

ComptrollerG2.sol
pragma solidity ^0.5.16;

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

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 */
contract ComptrollerG2 is ComptrollerV2Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    /**
     * @notice Emitted when an admin supports a market
     */
    event MarketListed(CToken cToken);

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

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

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

```

```

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

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

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

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

    /**
     * @notice Emitted when pause guardian is changed
     */
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

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

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

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

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

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

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

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

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

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

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

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

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

```

```

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);
            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }
        return results;
    }

    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];
        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }
        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }
        if (accountAssets[borrower].length >= maxAssets) {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }
        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        // this avoids having to iterate through the list for the most common use cases
        // that is, only when we need to perform liquidity checks
        // and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);
        emit MarketEntered(cToken, borrower);
        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     * or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code
        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return fail(Error.NONZERO_BORROW_BALANCE,
FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }
        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }
        Market storage marketToExit = markets[address(cToken)];
        /* Return true if the sender is not already 'in' the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }
        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];
        /* Delete cToken from the account's list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                userAssetList[i] = userAssetList[len - 1];
                len--;
            }
        }
        accountAssets[msg.sender] = userAssetList;
    }
}

```

```

for (uint i = 0; i < len; i++) {
    if (userAssetList[i] == cToken) {
        assetIndex = i;
        break;
    }
}
// We *must* have found the asset in the list or our redundant data structure is broken
assert(assetIndex < len);

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

emit MarketExited(cToken, msg.sender);

return uint(Error.NO_ERROR);
}

/** Policy Hooks ***/

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

    // Shh - currently unused
    minter;
    mintAmount;

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

    // *may include Policy Hook-type checks
    return uint(Error.NO_ERROR);
}

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

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

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

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

    // *may include Policy Hook-type checks
    /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
    if (!markets[cToken].accountMembership[redeemer]) {
}

```

```

        return uint(Error.NO_ERROR);
    }

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Validates redeem and reverts on rejection. May emit logs.
 * @param cToken Asset being redeemed
 * @param redeemer The address redeeming the tokens
 * @param redeemAmount The amount of the underlying asset being redeemed
 * @param redeemTokens The number of tokens being redeemed
 */
function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
    // Shh - currently unused
    cToken;
    redeemer;

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

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

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

    // *may include Policy Hook-type checks

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

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

    // it should be impossible to break the important invariant
    assert(markets[cToken].accountMembership[borrower]);

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Validates borrow and reverts on rejection. May emit logs.
 * @param cToken Asset whose underlying is being borrowed
 * @param borrower The address borrowing the underlying
 * @param borrowAmount The amount of the underlying asset requested to borrow
 */

```

```
function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
    // Shh - currently unused
    cToken;
    borrower;
    borrowAmount;

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

/**
 * @notice Checks if the account should be allowed to repay a borrow in the given market
 * @param cToken The market to verify the repay against
 * @param payer The account which would repay the asset
 * @param borrower The account which would borrowed the asset
 * @param repayAmount The amount of the underlying asset the account would repay
 * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function repayBorrowAllowed(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount) external returns (uint) {
    // Shh - currently unused
    payer;
    borrower;
    repayAmount;

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

    // *may include Policy Hook-type checks
    return uint(Error.NO_ERROR);
}

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

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

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

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

    // *may include Policy Hook-type checks
}
```

```

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

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

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

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

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

    // Shh - currently unused
    liquidator;
    borrower;
    seizeTokens;

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

    if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
        return uint(Error.COMPTROLLER_MISMATCH);
    }

    // *may include Policy Hook-type checks
    return uint(Error.NO_ERROR);
}

```

```


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

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

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint)

    {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // Shh - currently unused
        dst;

        // *may include Policy Hook-type checks

        // Currently the only consideration is whether or not
        // the src is allowed to redeem this many tokens
        return redeemAllowedInternal(cToken, src, transferTokens);
    }

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

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

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
     * Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     * whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
    }


```

```
        Exp oraclePrice;
        Exp tokensToEther;
    }

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

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

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
     *         hypothetical account liquidity in excess of collateral requirements,
     *         hypothetical account shortfall below collateral requirements)
    */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account,
CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

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

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

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

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account);
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
upgrades
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
            vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
            vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

            // Get the normalized price of the asset
            vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
            if (vars.oraclePriceMantissa == 0) {

```

```

        return (Error.PRICE_ERROR, 0, 0);
    }
    vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

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

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

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

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

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

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

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

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

    (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
}

```

```

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

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

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

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

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

/** Admin Functions **/


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

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

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sets the closeFactor used when liquidating borrows
 * @dev Admin function to set closeFactor
 * @param newCloseFactorMantissa New close factor, scaled by 1e18
 * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
 */
function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint256) {
    // Check caller is admin
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
    }

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

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

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

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sets the collateralFactor for a market
 * @dev Admin function to set per-market collateralFactor
 * @param cToken The market to set the factor on
 * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
 * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
 */
function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint256) {
    // Check caller is admin
}

```

```

if(msg.sender != admin) {
    return
}
FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
}                               fail(Error.UNAUTHORIZED,)

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

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

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

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

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

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

return uint(Error.NO_ERROR);
}

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

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

```

```

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

return uint(Error.NO_ERROR);
}

/**
 * @notice Add the market to the markets mapping and set it as listed
 * @dev Admin function to set isListed and add support for the market
 * @param cToken The address of the market (token) to list
 * @return uint 0=success, otherwise a failure. (See enum Error for details)
 */
function _supportMarket(CToken cToken) external returns (uint) {
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
    }

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

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

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

return uint(Error.NO_ERROR);
}

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

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

    // Store pauseGuardian with value newPauseGuardian
    pauseGuardian = newPauseGuardian;

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

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

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

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

```

```
require(msg.sender == admin || state == true, "only admin can unpause");

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

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

ComptrollerG3.sol

pragma solidity ^0.5.16;

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

/**
 * @title Compound's Comptroller Contract
 * @author Compound
 */
contract ComptrollerG3 is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
    ///@notice Emitted when an admin supports a market
    event MarketListed(CToken cToken);

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

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

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

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

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

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

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

    ///@notice Emitted when pause guardian is changed
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

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

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

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

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

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

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

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

    ///@notice The threshold above which the flywheel transfers COMP, in wei
}
```

```

uint public constant compClaimThreshold = 0.001e18;
/// @notice The initial COMP index for a market
uint224 public constant compInitialIndex = 1e36;
// closeFactorMantissa must be strictly greater than this value
uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05
// closeFactorMantissa must not exceed this value
uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9
// No collateralFactorMantissa may exceed this value
uint internal constant collateralFactorMaxMantissa = 0.9e18; // 0.9
// liquidationIncentiveMantissa must be no less than this value
uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0
// liquidationIncentiveMantissa must be no greater than this value
uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

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

/** Assets You Are In ***/

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

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

/**
 * @notice Add assets to be included in account liquidity calculation
 * @param cTokens The list of addresses of the cToken markets to be enabled
 * @return Success indicator for whether each corresponding market was entered
 */
function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
    uint len = cTokens.length;
    uint[] memory results = new uint[](len);
    for (uint i = 0; i < len; i++) {
        CToken cToken = CToken(cTokens[i]);
        results[i] = uint(addToMarketInternal(cToken, msg.sender));
    }
    return results;
}

/**
 * @notice Add the market to the borrower's "assets in" for liquidity calculations
 * @param cToken The market to enter
 * @param borrower The address of the account to modify
 * @return Success indicator for whether the market was entered
 */
function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
    Market storage marketToJoin = markets[address(cToken)];
    if (!marketToJoin.isListed) {
        // market is not listed, cannot join
        return Error.MARKET_NOT_LISTED;
    }
    if (marketToJoin.accountMembership[borrower] == true) {
        // already joined
        return Error.NO_ERROR;
    }
    if (accountAssets[borrower].length >= maxAssets) {
        // no space, cannot join
        return Error.TOO_MANY_ASSETS;
    }
}

```

```

        }

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

        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     * or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed,) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

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

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

        Market storage marketToExit = markets[address(cToken)];
        /* Return true if the sender is not already 'in' the market */
        if(!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

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

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

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

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

        emit MarketExited(cToken, msg.sender);

        return uint(Error.NO_ERROR);
    }

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

```

```

// Shh - currently unused
minter;
mintAmount;

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

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

return uint(Error.NO_ERROR);
}

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

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

return uint(Error.NO_ERROR);
}

/**
 * @notice Validates redeem and reverts on rejection. May emit logs.
 * @param cToken Asset being redeemed
 * @param redeemer The address redeeming the tokens
 * @param redeemAmount The amount of the underlying asset being redeemed
 * @param redeemTokens The number of tokens being redeemed
 */

```

```

function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
    // Shh - currently unused
    cToken;
    redeemer;

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

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

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

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

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

    // it should be impossible to break the important invariant
    assert(markets[cToken].accountMembership[borrower]);

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

/**
 * @notice Checks if the account should be allowed to repay a borrow in the given market
 * @param cToken The market to verify the repay against
 * @param payer The account which would repay the asset
 * @param borrower The account which would borrowed the asset
 * @param repayAmount The amount of the underlying asset the account would repay
 * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */

```

```

function repayBorrowAllowed(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount) external returns (uint) {
    // Shh - currently unused
    payer;
    borrower;
    repayAmount;

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

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

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

```

```

        }

        return uint(Error.NO_ERROR);
    }

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

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

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

        // Shh - currently unused
        seizeTokens;

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

        if(CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
            return uint(Error.COMPTROLLER_MISMATCH);
        }

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

        return uint(Error.NO_ERROR);
    }

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

```

```

liquidator;
borrower;
seizeTokens;

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

/**
 * @notice Checks if the account should be allowed to transfer tokens in the given market
 * @param cToken The market to verify the transfer against
 * @param src The account which sources the tokens
 * @param dst The account which receives the tokens
 * @param transferTokens The number of cTokens to transfer
 * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
 */
function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint)
{
    // Pausing is a very serious situation - we revert to sound the alarms
    require(!transferGuardianPaused, "transfer is paused");

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

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

    return uint(Error.NO_ERROR);
}

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

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

/** Liquidity/Liquidation Calculations ***/

/**
 * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
 * Note that `cTokenBalance` is the number of cTokens the account owns in the market,
 * whereas `borrowBalance` is the amount of underlying that the account has borrowed.
 */
struct AccountLiquidityLocalVars {
    uint sumCollateral;
    uint sumBorrowPlusEffects;
    uint cTokenBalance;
    uint borrowBalance;
    uint exchangeRateMantissa;
    uint oraclePriceMantissa;
    Exp collateralFactor;
    Exp exchangeRate;
    Exp oraclePrice;
    Exp tokensToDenom;
}

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

```

```

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

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

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
     *         hypothetical account liquidity in excess of collateral requirements,
     *         hypothetical account shortfall below collateral requirements)
    */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account,
CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

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

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

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

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account);
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
upgrades
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
            vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
            vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

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

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

            // sumCollateral += tokensToDenom * cTokenBalance
            (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom,

```

```

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

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

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

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

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

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

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

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

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

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

```

```

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

/*** Admin Functions ***/

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

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

```

```

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

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

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

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

return uint(Error.NO_ERROR);
}

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

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

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

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

    return uint(Error.NO_ERROR);
}

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

```

```
if(msg.sender != admin) {
    return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
}

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

cToken.isCToken(); // Sanity check to make sure its really a CToken
markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});
_addMarketInternal(address(cToken));
emit MarketListed(cToken);
return uint(Error.NO_ERROR);
}

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

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

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

    // Store pauseGuardian with value newPauseGuardian
    pauseGuardian = newPauseGuardian;

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

    return uint(Error.NO_ERROR);
}

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

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

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

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

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

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

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

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

```

        return state;
    }

    function become(Unitroller unitroller, uint compRate_, address[] memory compMarketsToAdd, address[] memory otherMarketsToAdd) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        require(unitroller._acceptImplementation() == 0, "change not authorized");

        ComptrollerG3(address(unitroller))._becomeG3(compRate_, compMarketsToAdd, otherMarketsToAdd);
    }

    function becomeG3(uint compRate_, address[] memory compMarketsToAdd, address[] memory otherMarketsToAdd) public {
        require(msg.sender == comptrollerImplementation, "only brains can become itself");

        for (uint i = 0; i < compMarketsToAdd.length; i++) {
            _addMarketInternal(address(compMarketsToAdd[i]));
        }

        for (uint i = 0; i < otherMarketsToAdd.length; i++) {
            _addMarketInternal(address(otherMarketsToAdd[i]));
        }

        _setCompRate(compRate_);
        _addCompMarkets(compMarketsToAdd);
    }

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

    /*** Comp Distribution ***/

    /**
     * @notice Recalculate and update COMP speeds for all COMP markets
     */
    function refreshCompSpeeds() public {
        CToken[] memory allMarkets_ = allMarkets;

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

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

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

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

```

```

        index: safe224(index.mantissa, "new index exceeds 224 bits"),
        block: safe32(blockNumber, "block number exceeds 32 bits")
    });
} else if(deltaBlocks > 0) {
    supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
}
}

/**
 * @notice Accrue COMP to the market by updating the borrow index
 * @param cToken The market whose borrow index to update
 */
function updateCompBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
    CompMarketState storage borrowState = compBorrowState[cToken];
    uint borrowSpeed = compSpeeds[cToken];
    uint blockNumber = getBlockNumber();
    uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
    if(deltaBlocks > 0 && borrowSpeed > 0) {
        uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
        uint compAccrued = mul_(deltaBlocks, borrowSpeed);
        Double memory ratio = borrowAmount > 0 ? fraction(compAccrued, borrowAmount) : Double({mantissa: 0});
        Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
        borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
    }
}

/**
 * @notice Calculate COMP accrued by a supplier and possibly transfer it to them
 * @param cToken The market in which the supplier is interacting
 * @param supplier The address of the supplier to distribute COMP to
 */
function distributeSupplierComp(address cToken, address supplier, bool distributeAll) internal {
    CompMarketState storage supplyState = compSupplyState[cToken];
    Double memory supplyIndex = Double({mantissa: supplyState.index});
    Double memory supplierIndex = Double({mantissa: compSupplierIndex[cToken][supplier]});
    compSupplierIndex[cToken][supplier] = supplyIndex.mantissa;
    if(supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
        supplierIndex.mantissa = compInitialIndex;
    }
    Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
    uint supplierTokens = CToken(cToken).balanceOf(supplier);
    uint supplierDelta = mul_(supplierTokens, deltaIndex);
    uint supplierAccrued = add_(compAccrued[supplier], supplierDelta);
    compAccrued[supplier] = transferComp(supplier, supplierAccrued, distributeAll ? 0 : compClaimThreshold);
    emit DistributedSupplierComp(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
}

/**
 * @notice Calculate COMP accrued by a borrower and possibly transfer it to them
 * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
 * @param cToken The market in which the borrower is interacting
 * @param borrower The address of the borrower to distribute COMP to
 */
function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex, bool distributeAll) internal {
    CompMarketState storage borrowState = compBorrowState[cToken];
    Double memory borrowIndex = Double({mantissa: borrowState.index});
    Double memory borrowerIndex = Double({mantissa: compBorrowerIndex[cToken][borrower]});
    compBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;
    if(borrowerIndex.mantissa > 0) {
        Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
        uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowIndex);
        uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
        uint borrowerAccrued = add_(compAccrued[borrower], borrowerDelta);
        compAccrued[borrower] = transferComp(borrower, borrowerAccrued, distributeAll ? 0 : compClaimThreshold);
        emit DistributedBorrowerComp(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
    }
}

/**
 * @notice Transfer COMP to the user, if they are above the threshold
 * @dev Note: If there is not enough COMP, we do not perform the transfer all.
 * @param user The address of the user to transfer COMP to
 * @param userAccrued The amount of COMP to (possibly) transfer
 */

```

```

    * @return The amount of COMP which was NOT transferred to the user
    */
function transferComp(address user, uint userAccrued, uint threshold) internal returns (uint) {
    if(userAccrued >= threshold && userAccrued > 0) {
        Comp comp = Comp(getCompAddress());
        uint compRemaining = comp.balanceOf(address(this));
        if(userAccrued <= compRemaining) {
            comp.transfer(user, userAccrued);
            return 0;
        }
    }
    return userAccrued;
}

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

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

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

/** Comp Distribution Admin ***/

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

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

    refreshCompSpeeds();
}

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

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

```

```

        refreshCompSpeeds();
    }

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

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

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

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

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

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

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

        refreshCompSpeeds();
    }

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

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

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

```

### ComptrollerInterface.sol

```

pragma solidity ^0.5.16;

contract ComptrollerInterface {
    ///@notice Indicator that this is a Comptroller contract (for inspection)
    bool public constant isComptroller = true;

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

    function enterMarkets(address[] calldata cTokens) external returns (uint[] memory);
    function exitMarket(address cToken) external returns (uint);

    /*** Policy Hooks ***/

    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint);
    function mintVerify(address cToken, address minter, uint mintAmount, uint mintTokens) external;

    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uint);
    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external;
}

```

```
function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint);
function borrowVerify(address cToken, address borrower, uint borrowAmount) external;

function repayBorrowAllowed(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount) external returns (uint);
function repayBorrowVerify(
    address cToken,
    address payer,
    address borrower,
    uint repayAmount,
    uint borrowerIndex) external;

function liquidateBorrowAllowed(
    address cTokenBorrowed,
    address cTokenCollateral,
    address liquidator,
    address borrower,
    uint repayAmount) external returns (uint);
function liquidateBorrowVerify(
    address cTokenBorrowed,
    address cTokenCollateral,
    address liquidator,
    address borrower,
    uint repayAmount,
    uint seizeTokens) external;

function seizeAllowed(
    address cTokenCollateral,
    address cTokenBorrowed,
    address liquidator,
    address borrower,
    uint seizeTokens) external returns (uint);
function seizeVerify(
    address cTokenCollateral,
    address cTokenBorrowed,
    address liquidator,
    address borrower,
    uint seizeTokens) external;

function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint);
function transferVerify(address cToken, address src, address dst, uint transferTokens) external;

/** Liquidity/Liquidation Calculations **/
function liquidateCalculateSeizeTokens(
    address cTokenBorrowed,
    address cTokenCollateral,
    uint repayAmount) external view returns (uint, uint);
}

ComptrollerStorage.sol
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./PriceOracle.sol";

contract UnitrollerAdminStorage {
    /**
     * @notice Administrator for this contract
     */
    address public admin;

    /**
     * @notice Pending administrator for this contract
     */
    address public pendingAdmin;

    /**
     * @notice Active brains of Unitroller
     */
    address public comptrollerImplementation;

    /**
     * @notice Pending brains of Unitroller
     */
    address public pendingComptrollerImplementation;
}

contract ComptrollerV1Storage is UnitrollerAdminStorage {
    /**

```

```
* @notice Oracle which gives the price of any given asset
*/
PriceOracle public oracle;

/**
 * @notice Multiplier used to calculate the maximum repayAmount when liquidating a borrow
 */
uint public closeFactorMantissa;

/**
 * @notice Multiplier representing the discount on collateral that a liquidator receives
 */
uint public liquidationIncentiveMantissa;

/**
 * @notice Max number of assets a single account can participate in (borrow or use as collateral)
 */
uint public maxAssets;

/**
 * @notice Per-account mapping of "assets you are in", capped by maxAssets
 */
mapping(address => CToken[]) public accountAssets;
}

contract ComptrollerV2Storage is ComptrollerV1Storage {
    struct Market {
        /**
         * @notice Whether or not this market is listed
         */
        bool isListed;

        /**
         * @notice Multiplier representing the most one can borrow against their collateral in this market.
         * For instance, 0.9 to allow borrowing 90% of collateral value.
         * Must be between 0 and 1, and stored as a mantissa.
         */
        uint collateralFactorMantissa;

        /**
         * @notice Per-market mapping of "accounts in this asset"
         */
        mapping(address => bool) accountMembership;

        /**
         * @notice Whether or not this market receives COMP
         */
        bool isComped;
    }

    /**
     * @notice Official mapping of cTokens -> Market metadata
     * @dev Used e.g. to determine if a market is supported
     */
    mapping(address => Market) public markets;

    /**
     * @notice The Pause Guardian can pause certain actions as a safety mechanism.
     * Actions which allow users to remove their own assets cannot be paused.
     * Liquidation / seizing / transfer can only be paused globally, not by market.
     */
    address public pauseGuardian;
    bool public mintGuardianPaused;
    bool public borrowGuardianPaused;
    bool public transferGuardianPaused;
    bool public seizeGuardianPaused;
    mapping(address => bool) public mintGuardianPaused;
    mapping(address => bool) public borrowGuardianPaused;
}

contract ComptrollerV3Storage is ComptrollerV2Storage {
    struct CompMarketState {
        /**
         * @notice The market's last updated compBorrowIndex or compSupplyIndex
         */
        uint224 index;

        /**
         * @notice The block number the index was last updated at
         */
        uint32 block;
    }

    /**
     * @notice A list of all markets
     */
    CToken[] public allMarkets;

    /**
     * @notice The rate at which the flywheel distributes COMP, per block
     */
    uint public compRate;

    /**
     * @notice The portion of compRate that each market currently receives
     */
    mapping(address => uint) public compSpeeds;

    /**
     * @notice The COMP market supply state for each market
     */
    mapping(address => CompMarketState) public compSupplyState;
}
```

```

///@notice The COMP market borrow state for each market
mapping(address => CompMarketState) public compBorrowState;

///@notice The COMP borrow index for each market for each supplier as of the last time they accrued
COMP
mapping(address => mapping(address => uint)) public compSupplierIndex;

///@notice The COMP borrow index for each market for each borrower as of the last time they accrued
COMP
mapping(address => mapping(address => uint)) public compBorrowerIndex;

///@notice The COMP accrued but not yet transferred to each user
mapping(address => uint) public compAccrued;
}

CToken.sol

pragma solidity ^0.5.16;

import "./ComptrollerInterface.sol";
import "./CTokenInterfaces.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./EIP20Interface.sol";
import "./EIP20NonStandardInterface.sol";
import "./InterestRateModel.sol";

/**
 * @title Compound's CToken Contract
 * @notice Abstract base for CTokens
 * @author Compound
 */
contract CToken is CTokenInterface, Exponential, TokenErrorReporter {
    /**
     * @notice Initialize the money market
     * @param comptroller_ The address of the Comptroller
     * @param interestRateModel_ The address of the interest rate model
     * @param initialExchangeRateMantissa_ The initial exchange rate, scaled by 1e18
     * @param name_ EIP-20 name of this token
     * @param symbol_ EIP-20 symbol of this token
     * @param decimals_ EIP-20 decimal precision of this token
     */
    function initialize(ComptrollerInterface comptroller,
                        InterestRateModel interestRateModel_,
                        uint initialExchangeRateMantissa_,
                        string memory name_,
                        string memory symbol_,
                        uint8 decimals_) public {
        require(msg.sender == admin, "only admin may initialize the market");
        require(accrualBlockNumber == 0 && borrowIndex == 0, "market may only be initialized once");

        // Set initial exchange rate
        initialExchangeRateMantissa = initialExchangeRateMantissa_;
        require(initialExchangeRateMantissa > 0, "initial exchange rate must be greater than zero.");

        // Set the comptroller
        uint err = _setComptroller(comptroller_);
        require(err == uint(Error.NO_ERROR), "setting comptroller failed");

        // Initialize block number and borrow index (block number mocks depend on comptroller being set)
        accrualBlockNumber = getBlockNumber();
        borrowIndex = mantissaOne;

        // Set the interest rate model (depends on block number / borrow index)
        err = _setInterestRateModelFresh(interestRateModel_);
        require(err == uint(Error.NO_ERROR), "setting interest rate model failed");

        name = name_;
        symbol = symbol_;
        decimals = decimals_;

        // The counter starts true to prevent changing it from zero to non-zero (i.e. smaller cost/refund)
        _notEntered = true;
    }

    /**
     * @notice Transfer `tokens` tokens from `src` to `dst` by `spender`
     * @dev Called by both `transfer` and `transferFrom` internally
     * @param spender The address of the account performing the transfer
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param tokens The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferTokens(address spender, address src, address dst, uint tokens) internal returns (uint) {
        /* Fail if transfer not allowed */
    }
}

```

```

uint allowed = comptroller.transferAllowed(address(this), src, dst, tokens);
if (allowed != 0) {
    return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.TRANSFER_COMPTROLLER_REJECTION, allowed);
}

/* Do not allow self-transfers */
if (src == dst) {
    return fail(Error.BAD_INPUT, FailureInfo.TRANSFER_NOT_ALLOWED);
}

/* Get the allowance, infinite for the account owner */
uint startingAllowance = 0;
if (spender == src) {
    startingAllowance = uint(-1);
} else {
    startingAllowance = transferAllowances[src][spender];
}

/* Do the calculations, checking for {under,over}flow */
MathError mathErr;
uint allowanceNew;
uint srcTokensNew;
uint dstTokensNew;

(mathErr, allowanceNew) = subUInt(startingAllowance, tokens);
if (mathErr != MathError.NO_ERROR) {
    return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ALLOWED);
}

(mathErr, srcTokensNew) = subUInt(accountTokens[src], tokens);
if (mathErr != MathError.NO_ERROR) {
    return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_NOT_ENOUGH);
}

(mathErr, dstTokensNew) = addUInt(accountTokens[dst], tokens);
if (mathErr != MathError.NO_ERROR) {
    return fail(Error.MATH_ERROR, FailureInfo.TRANSFER_TOO_MUCH);
}

/////////////////////////////
// EFFECTS & INTERACTIONS
// (No safe failures beyond this point)

accountTokens[src] = srcTokensNew;
accountTokens[dst] = dstTokensNew;

/* Eat some of the allowance (if necessary) */
if (startingAllowance != uint(-1)) {
    transferAllowances[src][spender] = allowanceNew;
}

/* We emit a Transfer event */
emit Transfer(src, dst, tokens);

comptroller.transferVerify(address(this), src, dst, tokens);
return uint(Error.NO_ERROR);
}

/**
 * @notice Transfer `amount` tokens from `msg.sender` to `dst`
 * @param dst The address of the destination account
 * @param amount The number of tokens to transfer
 * @return Whether or not the transfer succeeded
 */
function transfer(address dst, uint256 amount) external nonReentrant returns (bool) {
    return transferTokens(msg.sender, msg.sender, dst, amount) == uint(Error.NO_ERROR);
}

/**
 * @notice Transfer `amount` tokens from `src` to `dst`
 * @param src The address of the source account
 * @param dst The address of the destination account
 * @param amount The number of tokens to transfer
 * @return Whether or not the transfer succeeded
 */
function transferFrom(address src, address dst, uint256 amount) external nonReentrant returns (bool) {
    return transferTokens(msg.sender, src, dst, amount) == uint(Error.NO_ERROR);
}

/**
 * @notice Approve `spender` to transfer up to `amount` from `src`
 * @dev This will overwrite the approval amount for `spender`.
 * and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
 * @param spender The address of the account which may transfer tokens
 * @param amount The number of tokens that are approved (-1 means infinite)
 */

```

```

    * @return Whether or not the approval succeeded
    */
function approve(address spender, uint256 amount) external returns (bool) {
    address src = msg.sender;
    transferAllowances[src][spender] = amount;
    emit Approval(src, spender, amount);
    return true;
}

/**
    * @notice Get the current allowance from `owner` for `spender`
    * @param owner The address of the account which owns the tokens to be spent
    * @param spender The address of the account which may transfer tokens
    * @return The number of tokens allowed to be spent (-1 means infinite)
    */
function allowance(address owner, address spender) external view returns (uint256) {
    return transferAllowances[owner][spender];
}

/**
    * @notice Get the token balance of the `owner`
    * @param owner The address of the account to query
    * @return The number of tokens owned by `owner`
    */
function balanceOf(address owner) external view returns (uint256) {
    return accountTokens[owner];
}

/**
    * @notice Get the underlying balance of the `owner`
    * @dev This also accrues interest in a transaction
    * @param owner The address of the account to query
    * @return The amount of underlying owned by `owner`
    */
function balanceOfUnderlying(address owner) external returns (uint) {
    Exp memory exchangeRate = Exp({mantissa: exchangeRateCurrent()});
    (MathError mErr, uint balance) = mulScalarTruncate(exchangeRate, accountTokens[owner]);
    require(mErr == MathError.NO_ERROR, "balance could not be calculated");
    return balance;
}

/**
    * @notice Get a snapshot of the account's balances, and the cached exchange rate
    * @dev This is used by comptroller to more efficiently perform liquidity checks.
    * @param account Address of the account to snapshot
    * @return (possible error, token balance, borrow balance, exchange rate mantissa)
    */
function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint) {
    uint cTokenBalance = accountTokens[account];
    uint borrowBalance;
    uint exchangeRateMantissa;

    MathError mErr;
    (mErr, borrowBalance) = borrowBalanceStoredInternal(account);
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }

    (mErr, exchangeRateMantissa) = exchangeRateStoredInternal();
    if (mErr != MathError.NO_ERROR) {
        return (uint(Error.MATH_ERROR), 0, 0, 0);
    }

    return (uint(Error.NO_ERROR), cTokenBalance, borrowBalance, exchangeRateMantissa);
}

/**
    * @dev Function to simply retrieve block number
    * This exists mainly for inheriting test contracts to stub this result.
    */
function getBlockNumber() internal view returns (uint) {
    return block.number;
}

/**
    * @notice Returns the current per-block borrow interest rate for this cToken
    * @return The borrow interest rate per block, scaled by 1e18
    */
function borrowRatePerBlock() external view returns (uint) {
    return interestRateModel.getBorrowRate(getCashPrior(), totalBorrows, totalReserves);
}

/**
    * @notice Returns the current per-block supply interest rate for this cToken
    * @return The supply interest rate per block, scaled by 1e18
    */

```

```

function supplyRatePerBlock() external view returns (uint) {
    return interestRateModel.getSupplyRate(getCashPrior(), totalBorrows, totalReserves,
reserveFactorMantissa);
}

/**
 * @notice Returns the current total borrows plus accrued interest
 * @return The total borrows with interest
 */
function totalBorrowsCurrent() external nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return totalBorrows;
}

/**
 * @notice Accrue interest to updated borrowIndex and then calculate account's borrow balance using the
 * updated borrowIndex
 * @param account The address whose balance should be calculated after updating borrowIndex
 * @return The calculated balance
 */
function borrowBalanceCurrent(address account) external nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return borrowBalanceStored(account);
}

/**
 * @notice Return the borrow balance of account based on stored data
 * @param account The address whose balance should be calculated
 * @return The calculated balance
 */
function borrowBalanceStored(address account) public view returns (uint) {
    (MathError err, uint result) = borrowBalanceStoredInternal(account);
    require(err == MathError.NO_ERROR, "borrowBalanceStored: borrowBalanceStoredInternal failed");
    return result;
}

/**
 * @notice Return the borrow balance of account based on stored data
 * @param account The address whose balance should be calculated
 * @return (error code, the calculated balance or 0 if error code is non-zero)
 */
function borrowBalanceStoredInternal(address account) internal view returns (MathError, uint) {
    /* Note: we do not assert that the market is up to date */
    MathError mathErr;
    uint principalTimesIndex;
    uint result;

    /* Get borrowBalance and borrowIndex */
    BorrowSnapshot storage borrowSnapshot = accountBorrows[account];

    /* If borrowBalance = 0 then borrowIndex is likely also 0.
     * Rather than failing the calculation with a division by 0, we immediately return 0 in this case.
     */
    if(borrowSnapshot.principal == 0) {
        return (MathError.NO_ERROR, 0);
    }

    /* Calculate new borrow balance using the interest index:
     * recentBorrowBalance = borrower.borrowBalance * market.borrowIndex / borrower.borrowIndex
     */
    (mathErr, principalTimesIndex) = mulUInt(borrowSnapshot.principal, borrowIndex);
    if(mathErr != MathError.NO_ERROR) {
        return (mathErr, 0);
    }

    (mathErr, result) = divUInt(principalTimesIndex, borrowSnapshot.interestIndex);
    if(mathErr != MathError.NO_ERROR) {
        return (mathErr, 0);
    }

    return (MathError.NO_ERROR, result);
}

/**
 * @notice Accrue interest then return the up-to-date exchange rate
 * @return Calculated exchange rate scaled by 1e18
 */
function exchangeRateCurrent() public nonReentrant returns (uint) {
    require(accrueInterest() == uint(Error.NO_ERROR), "accrue interest failed");
    return exchangeRateStored();
}

/**
 * @notice Calculates the exchange rate from the underlying to the CToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return Calculated exchange rate scaled by 1e18
 */

```

```

function exchangeRateStored() public view returns (uint) {
    (MathError err, uint result) = exchangeRateStoredInternal();
    require(err == MathError.NO_ERROR, "exchangeRateStored: exchangeRateStoredInternal failed");
    return result;
}

/**
 * @notice Calculates the exchange rate from the underlying to the CToken
 * @dev This function does not accrue interest before calculating the exchange rate
 * @return (error code, calculated exchange rate scaled by 1e18)
 */
function exchangeRateStoredInternal() internal view returns (MathError, uint) {
    uint _totalSupply = totalSupply;
    if (_totalSupply == 0) {
        /*
         * If there are no tokens minted:
         * exchangeRate = initialExchangeRate
         */
        return (MathError.NO_ERROR, initialExchangeRateMantissa);
    } else {
        /*
         * Otherwise:
         * exchangeRate = (totalCash + totalBorrows - totalReserves) / totalSupply
         */
        uint totalCash = getCashPrior();
        uint cashPlusBorrowsMinusReserves;
        Exp memory exchangeRate;
        MathError mathErr;

        (mathErr, cashPlusBorrowsMinusReserves) = addThenSubUInt(totalCash, totalBorrows,
totalReserves);
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        (mathErr, exchangeRate) = getExp(cashPlusBorrowsMinusReserves, _totalSupply);
        if (mathErr != MathError.NO_ERROR) {
            return (mathErr, 0);
        }

        return (MathError.NO_ERROR, exchangeRate.mantissa);
    }
}

/**
 * @notice Get cash balance of this cToken in the underlying asset
 * @return The quantity of underlying asset owned by this contract
 */
function getCash() external view returns (uint) {
    return getCashPrior();
}

/**
 * @notice Applies accrued interest to total borrows and reserves
 * @dev This calculates interest accrued from the last checkpointed block
 *      up to the current block and writes new checkpoint to storage.
 */
function accrueInterest() public returns (uint) {
    /* Remember the initial block number */
    uint currentBlockNumber = getBlockNumber();
    uint accrualBlockNumberPrior = accrualBlockNumber;

    /* Short-circuit accumulating 0 interest */
    if (accrualBlockNumberPrior == currentBlockNumber) {
        return uint(Error.NO_ERROR);
    }

    /* Read the previous values out of storage */
    uint cashPrior = getCashPrior();
    uint borrowsPrior = totalBorrows;
    uint reservesPrior = totalReserves;
    uint borrowIndexPrior = borrowIndex;

    /* Calculate the current borrow interest rate */
    uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior, reservesPrior);
    require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");

    /* Calculate the number of blocks elapsed since the last accrual */
    (MathError mathErr, uint blockDelta) = subUInt(currentBlockNumber, accrualBlockNumberPrior);
    require(mathErr == MathError.NO_ERROR, "could not calculate block delta");

    /*
     * Calculate the interest accumulated into borrows and reserves and the new index:
     * simpleInterestFactor = borrowRate * blockDelta
     * interestAccumulated = simpleInterestFactor * totalBorrows
     * totalBorrowsNew = interestAccumulated + totalBorrows
     * totalReservesNew = interestAccumulated * reserveFactor + totalReserves
     */
}

```

```

    * borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
    */

    Exp memory simpleInterestFactor;
    uint interestAccumulated;
    uint totalBorrowsNew;
    uint totalReservesNew;
    uint borrowIndexNew;

    (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa: borrowRateMantissa}), blockDelta);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(Exp({mantissa: reserveFactorMantissa}),
interestAccumulated, reservesPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor, borrowIndexPrior,
borrowIndexPrior);
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED, uint(mathErr));
    }

    ///////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /* We write the previously calculated values into storage */
    accrualBlockNumber = currentBlockNumber;
    borrowIndex = borrowIndexNew;
    totalBorrows = totalBorrowsNew;
    totalReserves = totalReservesNew;

    /* We emit an AccrueInterest event */
    emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew);

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sender supplies assets into the market and receives cTokens in exchange
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param mintAmount The amount of the underlying asset to supply
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
mint amount.
*/
function mintInternal(uint mintAmount) internal nonReentrant returns (uint, uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow
failed
        return (fail(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED, 0);
    }
    // mintFresh emits the actual Mint event if successful and logs on errors, so we don't need to
return mintFresh(msg.sender, mintAmount);
}

struct MintLocalVars {
    Error err;
    MathError mathErr;
    uint exchangeRateMantissa;
    uint mintTokens;
    uint totalSupplyNew;
    uint accountTokensNew;
    uint actualMintAmount;
}

/**
 * @notice User supplies assets into the market and receives cTokens in exchange

```

```

    * @dev Assumes interest has already been accrued up to the current block
    * @param minter The address of the account which is supplying the assets
    * @param mintAmount The amount of the underlying asset to supply
    * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
mint amount.
*/
function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
    /* Fail if mint not allowed */
    uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);
    if (allowed != 0) {
        return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.MINT_COMPTROLLER_REJECTION, allowed), 0);
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
    }

    MintLocalVars memory vars;
    (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
    if (vars.mathErr != MathError.NO_ERROR) {
        return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED,
uint(vars.mathErr)), 0);
    }

    /////////////////
// EFFECTS & INTERACTIONS
// (No safe failures beyond this point)

/*
 * We call `doTransferIn` for the minter and the mintAmount.
 * Note: The cToken must handle variations between ERC-20 and ETH underlying.
 * `doTransferIn` reverts if anything goes wrong, since we can't be sure if
 * side-effects occurred. The function returns the amount actually transferred,
 * in case of a fee. On success, the cToken holds an additional `actualMintAmount`
 * of cash.
*/
vars.actualMintAmount = doTransferIn(minter, mintAmount);

/*
 * We get the current exchange rate and calculate the number of cTokens to be minted:
 * mintTokens = actualMintAmount / exchangeRate
*/
(vars.mathErr, vars.mintTokens) = divScalarByExpTruncate(vars.actualMintAmount, Exp({mantissa:
vars.exchangeRateMantissa}));
require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");

/*
 * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
 * totalSupplyNew = totalSupply + mintTokens
 * accountTokensNew = accountTokens[minter] + mintTokens
*/
(vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens);
require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");
(vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter], vars.mintTokens);
require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");

/* We write previously calculated values into storage */
totalSupply = vars.totalSupplyNew;
accountTokens[minter] = vars.accountTokensNew;

/* We emit a Mint event, and a Transfer event */
emit Mint(minter, vars.actualMintAmount, vars.mintTokens);
emit Transfer(address(this), minter, vars.mintTokens);

/* We call the defense hook */
comptroller.mintVerify(address(this), minter, vars.actualMintAmount, vars.mintTokens);

return (uint(Error.NO_ERROR), vars.actualMintAmount);
}

/**
 * @notice Sender redeems cTokens in exchange for the underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemTokens The number of cTokens to redeem into underlying
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function redeemInternal(uint redeemTokens) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem
}

```

```

failed
    return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
}
// redeemFresh emits redeem-specific logs on errors, so we don't need to
return redeemFresh(msg.sender, redeemTokens, 0);
}

/**
 * @notice Sender redeems cTokens in exchange for a specified amount of underlying asset
 * @dev Accrues interest whether or not the operation succeeds, unless reverted
 * @param redeemAmount The amount of underlying to receive from redeeming cTokens
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function redeemUnderlyingInternal(uint redeemAmount) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if(error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted redeem
failed
        return fail(Error(error), FailureInfo.REDEEM_ACCRUE_INTEREST_FAILED);
    }
    // redeemFresh emits redeem-specific logs on errors, so we don't need to
return redeemFresh(msg.sender, 0, redeemAmount);
}

struct RedeemLocalVars {
    Error err;
    MathError mathErr;
    uint exchangeRateMantissa;
    uint redeemTokens;
    uint redeemAmount;
    uint totalSupplyNew;
    uint accountTokensNew;
}

/**
 * @notice User redeems cTokens in exchange for the underlying asset
 * @dev Assumes interest has already been accrued up to the current block
 * @param redeemer The address of the account which is redeeming the tokens
 * @param redeemTokensIn The number of cTokens to redeem into underlying (only one of redeemTokensIn
or redeemAmountIn may be non-zero)
 * @param redeemAmountIn The number of underlying tokens to receive from redeeming cTokens (only one
of redeemTokensIn or redeemAmountIn may be non-zero)
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
*/
function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal
returns (uint) {
    require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn
must be zero");

    RedeemLocalVars memory vars;

    /* exchangeRate = invoke Exchange Rate Stored() */
    (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal();
    if(vars.mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr));
    }

    /* If redeemTokensIn > 0: */
    if(redeemTokensIn > 0) {
        /*
         * We calculate the exchange rate and the amount of underlying to be redeemed:
         * redeemTokens = redeemTokensIn
         * redeemAmount = redeemTokensIn x exchangeRateCurrent
        */
        vars.redeemTokens = redeemTokensIn;
        (vars.mathErr, vars.redeemAmount) = mulScalarTruncate(Exp({mantissa:
vars.exchangeRateMantissa}), redeemTokensIn);
        if(vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED, uint(vars.mathErr));
        }
    } else {
        /*
         * We get the current exchange rate and calculate the amount to be redeemed:
         * redeemTokens = redeemAmountIn / exchangeRate
         * redeemAmount = redeemAmountIn
        */
        (vars.mathErr, vars.redeemTokens) = divScalarByExpTruncate(redeemAmountIn, Exp({mantissa:
vars.exchangeRateMantissa}));
        if(vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED, uint(vars.mathErr));
        }
    }
}

```

```

        vars.redeemAmount = redeemAmountIn;
    }

    /* Fail if redeem not allowed */
    uint allowed = comptroller.redeemAllowed(address(this), redeemer, vars.redeemTokens);
    if(allowed != 0) {
        return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.REDEEM_COMPTROLLER_REJECTION, allowed);
    }

    /* Verify market's block number equals current block number */
    if(accrualBlockNumber != getBlockNumber()) {
        return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDEEM_FRESHNESS_CHECK);
    }

    /*
     * We calculate the new total supply and redeemer balance, checking for underflow:
     * totalSupplyNew = totalSupply - redeemTokens
     * accountTokensNew = accountTokens[redeemer] - redeemTokens
     */
    (vars.mathErr, vars.totalSupplyNew) = subUInt(totalSupply, vars.redeemTokens);
    if(vars.mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED, uint(vars.mathErr));
    }

    (vars.mathErr, vars.accountTokensNew) = subUInt(accountTokens[redeemer], vars.redeemTokens);
    if(vars.mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR,
FailureInfo.REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
    }

    /* Fail gracefully if protocol has insufficient cash */
    if(getCashPrior() < vars.redeemAmount) {
        return fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.REDEEM_TRANSFER_OUT_NOT_POSSIBLE);
    }

    ///////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    /*
     * We invoke doTransferOut for the redeemer and the redeemAmount.
     * Note: The cToken must handle variations between ERC-20 and ETH underlying.
     * On success, the cToken has redeemAmount less of cash.
     * doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
     */
    doTransferOut(redeemer, vars.redeemAmount);

    /* We write previously calculated values into storage */
    totalSupply = vars.totalSupplyNew;
    accountTokens[redeemer] = vars.accountTokensNew;

    /* We emit a Transfer event, and a Redeem event */
    emit Transfer(redeemer, address(this), vars.redeemTokens);
    emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);

    /* We call the defense hook */
    comptroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.redeemTokens);

    return uint(Error.NO_ERROR);
}

/**
 * @notice Sender borrows assets from the protocol to their own address
 * @param borrowAmount The amount of the underlying asset to borrow
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function borrowInternal(uint borrowAmount) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if(error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow
failed
        return fail(error, FailureInfo.BORROW_ACCRUE_INTEREST_FAILED);
    }
    // borrowFresh emits borrow-specific logs on errors, so we don't need to
    return borrowFresh(msg.sender, borrowAmount);
}

struct BorrowLocalVars {
    MathError mathErr;
    uint accountBorrows;
    uint accountBorrowsNew;
    uint totalBorrowsNew;
}

```

```


    /**
     * @notice Users borrow assets from the protocol to their own address
     * @param borrowAmount The amount of the underlying asset to borrow
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) {
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount);
        if(allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if(accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
        }

        /* Fail gracefully if protocol has insufficient underlying cash */
        if(getCashPrior() < borrowAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.BORROW_CASH_NOT_AVAILABLE);
        }

        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         * accountBorrowsNew = accountBorrows + borrowAmount
         * totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        if(vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.accountBorrowsNew) = addUIInt(vars.accountBorrows, borrowAmount);
        if(vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUIInt(totalBorrows, borrowAmount);
        if(vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        /////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         * Note: The cToken must handle variations between ERC-20 and ETH underlying.
         * On success, the cToken borrowAmount less of cash.
         * doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        doTransferOut(borrower, borrowAmount);

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a Borrow event */
        emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

        /* We call the defense hook */
        comptroller.borrowVerify(address(this), borrower, borrowAmount);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Sender repays their own borrow
     * @param repayAmount The amount to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
     * repayment amount.
     */
    function repayBorrowInternal(uint repayAmount) internal nonReentrant returns (uint, uint) {
        uint error = accrueInterest();
        if(error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow
failed
            return (fail(error), FailureInfo.REPAY_BORROW_ACCRUE_INTEREST_FAILED, 0);
        }
    }
}


```

```

        }  

        // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to  

        // return repayBorrowFresh(msg.sender, msg.sender, repayAmount);  

    }  

    /**
     * @notice Sender repays a borrow belonging to borrower
     * @param borrower the account with the debt being payed off
     * @param repayAmount The amount to repay
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
     repayment amount.
     */  

    function repayBorrowBehalfInternal(address borrower, uint repayAmount) internal nonReentrant returns
    (uint, uint) {  

        uint error = accrueInterest();  

        if(error != uint(Error.NO_ERROR)) {  

            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted borrow
            failed  

            return (fail(Error(error), FailureInfo.REPAY_BEHALF_ACCRUE_INTEREST_FAILED), 0);  

        }  

        // repayBorrowFresh emits repay-borrow-specific logs on errors, so we don't need to
        // return repayBorrowFresh(msg.sender, borrower, repayAmount);  

    }  

    struct RepayBorrowLocalVars {  

        Error err;  

        MathError mathErr;  

        uint repayAmount;  

        uint borrowerIndex;  

        uint accountBorrows;  

        uint accountBorrowsNew;  

        uint totalBorrowsNew;  

        uint actualRepayAmount;  

    }  

    /**
     * @notice Borrows are repaid by another user (possibly the borrower).
     * @param payer the account paying off the borrow
     * @param borrower the account with the debt being payed off
     * @param repayAmount the amount of underlying tokens being returned
     * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
     repayment amount.
     */  

    function repayBorrowFresh(address payer, address borrower, uint repayAmount) internal returns (uint, uint) {  

        /* Fail if repayBorrow not allowed */  

        uint allowed = comptroller.repayBorrowAllowed(address(this), payer, borrower, repayAmount);  

        if(allowed != 0) {  

            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION, allowed), 0);  

        }  

        /* Verify market's block number equals current block number */  

        if(accrualBlockNumber != getBlockNumber()) {  

            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.REPAY_BORROW_FRESHNESS_CHECK), 0);  

        }  

        RepayBorrowLocalVars memory vars;  

        /* We remember the original borrowerIndex for verification purposes */
        vars.borrowerIndex = accountBorrows[borrower].interestIndex;  

        /* We fetch the amount the borrower owes, with accumulated interest */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);  

        if(vars.mathErr != MathError.NO_ERROR) {  

            return (failOpaque(Error.MATH_ERROR, FailureInfo.REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr)), 0);  

        }  

        /* If repayAmount == -1, repayAmount = accountBorrows */  

        if(repayAmount == uint(-1)) {  

            vars.repayAmount = vars.accountBorrows;  

        } else {  

            vars.repayAmount = repayAmount;  

        }  

        ///////////////////////////////  

        // EFFECTS & INTERACTIONS  

        // (No safe failures beyond this point)  

        /*
         * We call doTransferIn for the payer and the repayAmount
         * Note: The cToken must handle variations between ERC-20 and ETH underlying.
         * On success, the cToken holds an additional repayAmount of cash.
         * doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
         * it returns the amount actually transferred, in case of a fee.
         */
    }

```

```

vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

/*
 * We calculate the new borrower and total borrow balances, failing on underflow:
 * accountBorrowsNew = accountBorrows - actualRepayAmount
 * totalBorrowsNew = totalBorrows - actualRepayAmount
 */
(vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.actualRepayAmount);
require(vars.mathErr == MathError.NO_ERROR,
"REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED");

(vars.mathErr, vars.totalBorrowsNew) = subUInt(totalBorrows, vars.actualRepayAmount);
require(vars.mathErr == MathError.NO_ERROR,
"REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED");

/* We write the previously calculated values into storage */
accountBorrows[borrower].principal = vars.accountBorrowsNew;
accountBorrows[borrower].interestIndex = borrowIndex;
totalBorrows = vars.totalBorrowsNew;

/* We emit a RepayBorrow event */
emit RepayBorrow(payer, borrower, vars.actualRepayAmount, vars.accountBorrowsNew,
vars.totalBorrowsNew);

/* We call the defense hook */
comptroller.repayBorrowVerify(address(this), payer, borrower, vars.actualRepayAmount,
vars.borrowerIndex);

return (uint(Error.NO_ERROR), vars.actualRepayAmount);
}

/**
 * @notice The sender liquidates the borrowers collateral.
 * The collateral seized is transferred to the liquidator.
 * @param borrower The borrower of this cToken to be liquidated
 * @param cTokenCollateral The market in which to seize collateral from the borrower
 * @param repayAmount The amount of the underlying borrowed asset to repay
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
repayment amount.
*/
function liquidateBorrowInternal(address borrower, uint repayAmount, CTokenInterface cTokenCollateral)
internal nonReentrant returns (uint, uint) {
    uint error = accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted liquidation
failed
        return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED), 0);
    }

    error = cTokenCollateral.accrueInterest();
    if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact that an attempted liquidation
failed
        return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED), 0);
    }

    // liquidateBorrowFresh emits borrow-specific logs on errors, so we don't need to
    return liquidateBorrowFresh(msg.sender, borrower, repayAmount, cTokenCollateral);
}

/**
 * @notice The liquidator liquidates the borrowers collateral.
 * The collateral seized is transferred to the liquidator.
 * @param borrower The borrower of this cToken to be liquidated
 * @param liquidator The address repaying the borrow and seizing collateral
 * @param cTokenCollateral The market in which to seize collateral from the borrower
 * @param repayAmount The amount of the underlying borrowed asset to repay
 * @return (uint, uint) An error code (0=success, otherwise a failure, see ErrorReporter.sol), and the actual
repayment amount.
*/
function liquidateBorrowFresh(address liquidator, address borrower, uint repayAmount, CTokenInterface
cTokenCollateral) internal returns (uint, uint) {
    /* Fail if liquidate not allowed */
    uint allowed = comptroller.liquidateBorrowAllowed(address(this), address(cTokenCollateral),
liquidator, borrower, repayAmount);
    if (allowed != 0) {
        return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
    }

    /* Verify cTokenCollateral market's block number equals current block number */
}

```

```

if(cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
    return
}
FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK), 0);
}

/* Fail if borrower = liquidator */
if(borrower == liquidator) {
    return
}
FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER), 0);

/* Fail if repayAmount = 0 */
if(repayAmount == 0) {
    return
}
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_ZERO), 0);

/* Fail if repayAmount = -1 */
if(repayAmount == uint(-1)) {
    return
}
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);

/* Fail if repayBorrow fails */
(uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower,
repayAmount);
if(repayBorrowError != uint(Error.NO_ERROR)) {
    return
}
FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED), 0);

/////////////////////////////
// EFFECTS & INTERACTIONS
// (No safe failures beyond this point)

/* We calculate the number of collateral tokens that will be seized */
(uint amountSeizeError, uint seizeTokens) = comptroller.liquidateCalculateSeizeTokens(address(this),
address(cTokenCollateral), actualRepayAmount);
require(amountSeizeError == uint(Error.NO_ERROR),
"LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");

/* Revert if borrower collateral token balance < seizeTokens */
require(cTokenCollateral.balanceOf(borrower) >= seizeTokens, "LIQUIDATE_SEIZE_TOO MUCH");

// If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make an external call
uint seizeError;
if(address(cTokenCollateral) == address(this)) {
    seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
} else {
    seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
}

/* Revert if seize tokens fails (since we cannot be sure of side effects) */
require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

/* We emit a LiquidateBorrow event */
emit LiquidateBorrow(liquidator, borrower, actualRepayAmount, address(cTokenCollateral),
seizeTokens);

/* We call the defense hook */
comptroller.liquidateBorrowVerify(address(this), address(cTokenCollateral), liquidator, borrower,
actualRepayAmount, seizeTokens);

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

/**
 * @notice Transfers collateral tokens (this market) to the liquidator.
 * @dev Will fail unless called by another cToken during the process of liquidation.
 * Its absolutely critical to use msg.sender as the borrowed cToken and not a parameter.
 * @param liquidator The account receiving seized collateral
 * @param borrower The account having collateral seized
 * @param seizeTokens The number of cTokens to seize
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function seize(address liquidator, address borrower, uint seizeTokens) external nonReentrant returns (uint) {
    return seizeInternal(msg.sender, liquidator, borrower, seizeTokens);
}

/**
 * @notice Transfers collateral tokens (this market) to the liquidator.
 * @dev Called only during an in-kind liquidation, or by liquidateBorrow during the liquidation of another
CToken.
 * Its absolutely critical to use msg.sender as the seizer cToken and not a parameter.
 * @param seizerToken The contract seizing the collateral (i.e. borrowed cToken)
 * @param liquidator The account receiving seized collateral
 */

```

```

    * @param borrower The account having collateral seized
    * @param seizeTokens The number of cTokens to seize
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
    function seizeInternal(address seizerToken, address liquidator, address borrower, uint seizeTokens) internal
    returns (uint);

    /**
     * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the
     * transfer.
     * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to
     * finalize the transfer.
     * @param newPendingAdmin New pending admin.
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _setPendingAdmin(address payable newPendingAdmin) external returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
     * @dev Admin function for pending admin to accept role and update admin
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _acceptAdmin() external returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin != address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }

        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        // Emit NewAdmin(oldAdmin, admin)
        emit NewAdmin(oldAdmin, admin);
        // Emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }

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

        ComptrollerInterface oldComptroller = comptroller;
        // Ensure invoke comptroller.isComptroller() returns true
        require(newComptroller.isComptroller(), "marker method returned false");

        // Set market's comptroller to newComptroller
        comptroller = newComptroller;

        // Emit NewComptroller(oldComptroller, newComptroller)
        emit NewComptroller(oldComptroller, newComptroller);

        return uint(Error.NO_ERROR);
    }
}

```

```

/*
 * @notice accrues interest and sets a new reserve factor for the protocol using _setReserveFactorFresh
 * @dev Admin function to accrue interest and set a new reserve factor
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _setReserveFactor(uint newReserveFactorMantissa) external nonReentrant returns (uint) {
    uint error = accrueInterest();
    if(error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but on top of that we want to log the fact that an attempted
        // reserve factor change failed.
        return fail(Error(error), FailureInfo.SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED);
    }
    // _setReserveFactorFresh emits reserve-factor-specific logs on errors, so we don't need to.
    return _setReserveFactorFresh(newReserveFactorMantissa);
}

/*
 * @notice Sets a new reserve factor for the protocol (*requires fresh interest accrual)
 * @dev Admin function to set a new reserve factor
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _setReserveFactorFresh(uint newReserveFactorMantissa) internal returns (uint) {
    // Check caller is admin
    if(msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_RESERVE_FACTOR_ADMIN_CHECK);
    }
    // Verify market's block number equals current block number
    if(accuralBlockNumber != getBlockNumber()) {
        return
            fail(Error.MARKET_NOT_FRESH,
FailureInfo.SET_RESERVE_FACTOR_FRESH_CHECK);
    }
    // Check newReserveFactor ≤ maxReserveFactor
    if(newReserveFactorMantissa > reserveFactorMaxMantissa) {
        return fail(Error.BAD_INPUT, FailureInfo.SET_RESERVE_FACTOR_BOUNDS_CHECK);
    }
    uint oldReserveFactorMantissa = reserveFactorMantissa;
    reserveFactorMantissa = newReserveFactorMantissa;
    emit NewReserveFactor(oldReserveFactorMantissa, newReserveFactorMantissa);
    return uint(Error.NO_ERROR);
}

/*
 * @notice Accrues interest and reduces reserves by transferring from msg.sender
 * @param addAmount Amount of addition to reserves
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _addReservesInternal(uint addAmount) internal nonReentrant returns (uint) {
    uint error = accrueInterest();
    if(error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but on top of that we want to log the fact that an attempted
        // reduce reserves failed.
        return fail(Error(error), FailureInfo.ADD_RESERVES_ACCRUE_INTEREST_FAILED);
    }
    // _addReservesFresh emits reserve-addition-specific logs on errors, so we don't need to.
    (error, ) = _addReservesFresh(addAmount);
    return error;
}

/*
 * @notice Add reserves by transferring from caller
 * @dev Requires fresh interest accrual
 * @param addAmount Amount of addition to reserves
 * @return (uint, uint) An error code (0=success, otherwise a failure (see ErrorReporter.sol for details)) and
the actual amount added, net token fees
 */
function _addReservesFresh(uint addAmount) internal returns (uint, uint) {
    // totalReserves + actualAddAmount
    uint totalReservesNew;
    uint actualAddAmount;

    // We fail gracefully unless market's block number equals current block number
    if(accuralBlockNumber != getBlockNumber()) {
        return
            (fail(Error.MARKET_NOT_FRESH, FailureInfo.ADD_RESERVES_FRESH_CHECK),
actualAddAmount);
    }

    /////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)
    /*

```

```

    * We call doTransferIn for the caller and the addAmount
    * Note: The cToken must handle variations between ERC-20 and ETH underlying.
    * On success, the cToken holds an additional addAmount of cash.
    * doTransferIn reverts if anything goes wrong, since we can't be sure if side effects occurred.
    * it returns the amount actually transferred, in case of a fee.
    */

    actualAddAmount = doTransferIn(msg.sender, addAmount);
    totalReservesNew = totalReserves + actualAddAmount;

    /* Revert on overflow */
    require(totalReservesNew >= totalReserves, "add reserves unexpected overflow");

    // Store reserves[n+1] = reserves[n] + actualAddAmount
    totalReserves = totalReservesNew;

    /* Emit NewReserves(admin, actualAddAmount, reserves[n+1]) */
    emit ReservesAdded(msg.sender, actualAddAmount, totalReservesNew);

    /* Return (NO_ERROR, actualAddAmount) */
    return (uint(Error.NO_ERROR), actualAddAmount);
}

/**
 * @notice Accrues interest and reduces reserves by transferring to admin
 * @param reduceAmount Amount of reduction to reserves
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _reduceReserves(uint reduceAmount) external nonReentrant returns (uint) {
    uint error = accrueInterest();
    if(error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but on top of that we want to log the fact that an attempted
        // reduce reserves failed.
        return fail(Error(error), FailureInfo.REDUCE_RESERVES_ACCRUE_INTEREST_FAILED);
    }
    // _reduceReservesFresh emits reserve-reduction-specific logs on errors, so we don't need to.
    return _reduceReservesFresh(reduceAmount);
}

/**
 * @notice Reduces reserves by transferring to admin
 * @dev Requires fresh interest accrual
 * @param reduceAmount Amount of reduction to reserves
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
    // totalReserves - reduceAmount
    uint totalReservesNew;

    // Check caller is admin
    if(msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.REDUCE_RESERVES_ADMIN_CHECK);
    }

    // We fail gracefully unless market's block number equals current block number
    if(accrualBlockNumber != getBlockNumber()) {
        return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDUCE_RESERVES_FRESH_CHECK);
    }

    // Fail gracefully if protocol has insufficient underlying cash
    if(getCashPrior() < reduceAmount) {
        return fail(Error.TOKEN_INSUFFICIENT_CASH,
FailureInfo.REDUCE_RESERVES_CASH_NOT_AVAILABLE);
    }

    // Check reduceAmount ≤ reserves[n] (totalReserves)
    if(reduceAmount > totalReserves) {
        return fail(Error.BAD_INPUT, FailureInfo.REDUCE_RESERVES_VALIDATION);
    }

    /////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    totalReservesNew = totalReserves - reduceAmount;
    // We checked reduceAmount <= totalReserves above, so this should never revert.
    require(totalReservesNew <= totalReserves, "reduce reserves unexpected underflow");

    // Store reserves[n+1] = reserves[n] - reduceAmount
    totalReserves = totalReservesNew;

    // doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
    doTransferOut(admin, reduceAmount);

    emit ReservesReduced(admin, reduceAmount, totalReservesNew);
}

```

```

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice accrues interest and updates the interest rate model using _setInterestRateModelFresh
     * @dev Admin function to accrue interest and update the interest rate model
     * @param newInterestRateModel the new interest rate model to use
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
        uint error = accrueInterest();
        if(error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but on top of that we want to log the fact that an attempted
            change of interest rate model failed
            return fail(Error(error),
FailureInfo.SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED));
        }
        // _setInterestRateModelFresh emits interest-rate-model-update-specific logs on errors, so we don't need
        to.
        return _setInterestRateModelFresh(newInterestRateModel);
    }

    /**
     * @notice updates the interest rate model (*requires fresh interest accrual)
     * @dev Admin function to update the interest rate model
     * @param newInterestRateModel the new interest rate model to use
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _setInterestRateModelFresh(InterestRateModel newInterestRateModel) internal returns (uint) {
        // Used to store old model for use in the event that is emitted on success
        InterestRateModel oldInterestRateModel;

        // Check caller is admin
        if(msg.sender != admin) {
            return fail(Error.UNAUTHORIZED,
FailureInfo.SET_INTEREST_RATE_MODEL_OWNER_CHECK);
        }

        // We fail gracefully unless market's block number equals current block number
        if(acrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH,
FailureInfo.SET_INTEREST_RATE_MODEL_FRESH_CHECK);
        }

        // Track the market's current interest rate model
        oldInterestRateModel = interestRateModel;

        // Ensure invoke newInterestRateModel.isInterestRateModel() returns true
        require(newInterestRateModel.isInterestRateModel(), "marker method returned false");

        // Set the interest rate model to newInterestRateModel
        interestRateModel = newInterestRateModel;

        // Emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel)
        emit NewMarketInterestRateModel(oldInterestRateModel, newInterestRateModel);

        return uint(Error.NO_ERROR);
    }

    /**
     * @notice Gets balance of this contract in terms of the underlying
     * @dev This excludes the value of the current message, if any
     * @return The quantity of underlying owned by this contract
     */
    function getCashPrior() internal view returns (uint);

    /**
     * @dev Performs a transfer in, reverting upon failure. Returns the amount actually transferred to the
     protocol, in case of a fee.
     * This may revert due to insufficient balance or insufficient allowance.
     */
    function doTransferIn(address from, uint amount) internal returns (uint);

    /**
     * @dev Performs a transfer out, ideally returning an explanatory error code upon failure rather than
     reverting.
     * If caller has not called checked protocol's balance, may revert due to insufficient cash held in the
     contract.
     * If caller has checked protocol's balance, and verified it is >= amount, this should not revert in normal
     conditions.
     */
    function doTransferOut(address payable to, uint amount) internal;
}

```

```
/** Reentrancy Guard */
/*
 * @dev Prevents a contract from calling itself, directly or indirectly.
 */
modifier nonReentrant() {
    require( _notEntered, "re-entered");
    _notEntered = false;
}
```
`_notEntered = true; // get a gas-refund post-Istanbul
}
```

*CTokenInterfaces.sol*

```
pragma solidity ^0.5.16;

import "./ComptrollerInterface.sol";
import "./InterestRateModel.sol";

contract CTokenStorage {
    /**
     * @dev Guard variable for re-entrancy checks
     */
    bool internal _notEntered;

    /**
     * @notice EIP-20 token name for this token
     */
    string public name;

    /**
     * @notice EIP-20 token symbol for this token
     */
    string public symbol;

    /**
     * @notice EIP-20 token decimals for this token
     */
    uint8 public decimals;

    /**
     * @notice Maximum borrow rate that can ever be applied (.0005% / block)
     */
    uint internal constant borrowRateMaxMantissa = 0.0005e16;

    /**
     * @notice Maximum fraction of interest that can be set aside for reserves
     */
    uint internal constant reserveFactorMaxMantissa = 1e18;

    /**
     * @notice Administrator for this contract
     */
    address payable public admin;

    /**
     * @notice Pending administrator for this contract
     */
    address payable public pendingAdmin;

    /**
     * @notice Contract which oversees inter-cToken operations
     */
    ComptrollerInterface public comptroller;

    /**
     * @notice Model which tells what the current interest rate should be
     */
    InterestRateModel public interestRateModel;

    /**
     * @notice Initial exchange rate used when minting the first CTokens (used when totalSupply = 0)
     */
    uint internal initialExchangeRateMantissa;

    /**
     * @notice Fraction of interest currently set aside for reserves
     */
    uint public reserveFactorMantissa;

    /**
     * @notice Block number that interest was last accrued at
     */
}
```

```
uint public accrualBlockNumber;

/**
 * @notice Accumulator of the total earned interest rate since the opening of the market
 */
uint public borrowIndex;

/**
 * @notice Total amount of outstanding borrows of the underlying in this market
 */
uint public totalBorrows;

/**
 * @notice Total amount of reserves of the underlying held in this market
 */
uint public totalReserves;

/**
 * @notice Total number of tokens in circulation
 */
uint public totalSupply;

/**
 * @notice Official record of token balances for each account
 */
mapping (address => uint) internal accountTokens;

/**
 * @notice Approved token transfer amounts on behalf of others
 */
mapping (address => mapping (address => uint)) internal transferAllowances;

/**
 * @notice Container for borrow balance information
 * @member principal Total balance (with accrued interest), after applying the most recent
balance-changing action
 * @member interestIndex Global borrowIndex as of the most recent balance-changing action
 */
struct BorrowSnapshot {
    uint principal;
    uint interestIndex;
}

/**
 * @notice Mapping of account addresses to outstanding borrow balances
 */
mapping(address => BorrowSnapshot) internal accountBorrows;
}

contract CTokenInterface is CTokenStorage {
    /**
     * @notice Indicator that this is a CToken contract (for inspection)
     */
    bool public constant isCToken = true;

    /**** Market Events ****/
    /**
     * @notice Event emitted when interest is accrued
     */
    event AccrueInterest(uint cashPrior, uint interestAccumulated, uint borrowIndex, uint totalBorrows);

    /**
     * @notice Event emitted when tokens are minted
     */
    event Mint(address minter, uint mintAmount, uint mintTokens);

    /**
     * @notice Event emitted when tokens are redeemed
     */
    event Redeem(address redeemer, uint redeemAmount, uint redeemTokens);

    /**
     * @notice Event emitted when underlying is borrowed
     */
    event Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows);

    /**
     * @notice Event emitted when a borrow is repaid
     */
    event RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint
totalBorrows);

    /**
     * @notice Event emitted when a borrow is liquidated
     */
}
```

```
event LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address cTokenCollateral, uint seizeTokens);

/***
 * @notice Event emitted when pendingAdmin is changed
 */
event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

/***
 * @notice Event emitted when pendingAdmin is accepted, which means admin is updated
 */
event NewAdmin(address oldAdmin, address newAdmin);

/***
 * @notice Event emitted when comptroller is changed
 */
event NewComptroller(ComptrollerInterface oldComptroller, ComptrollerInterface newComptroller);

/***
 * @notice Event emitted when interestRateModel is changed
 */
event NewMarketInterestRateModel(InterestRateModel oldInterestRateModel, InterestRateModel newInterestRateModel);

/***
 * @notice Event emitted when the reserve factor is changed
 */
event NewReserveFactor(uint oldReserveFactorMantissa, uint newReserveFactorMantissa);

/***
 * @notice Event emitted when the reserves are added
 */
event ReservesAdded(address benefactor, uint addAmount, uint newTotalReserves);

/***
 * @notice Event emitted when the reserves are reduced
 */
event ReservesReduced(address admin, uint reduceAmount, uint newTotalReserves);

/***
 * @notice EIP20 Transfer event
 */
event Transfer(address indexed from, address indexed to, uint amount);

/***
 * @notice EIP20 Approval event
 */
event Approval(address indexed owner, address indexed spender, uint amount);

/***
 * @notice Failure event
 */
event Failure(uint error, uint info, uint detail);

/***
 * User Interface
 */

function transfer(address dst, uint amount) external returns (bool);
function transferFrom(address src, address dst, uint amount) external returns (bool);
function approve(address spender, uint amount) external returns (bool);
function allowance(address owner, address spender) external view returns (uint);
function balanceOf(address owner) external view returns (uint);
function balanceOfUnderlying(address owner) external returns (uint);
function getAccountSnapshot(address account) external view returns (uint, uint, uint, uint);
function borrowRatePerBlock() external view returns (uint);
function supplyRatePerBlock() external view returns (uint);
function totalBorrowsCurrent() external returns (uint);
function borrowBalanceCurrent(address account) external returns (uint);
function borrowBalanceStored(address account) public view returns (uint);
function exchangeRateCurrent() public returns (uint);
function exchangeRateStored() public view returns (uint);
function getCash() external view returns (uint);
function accrueInterest() public returns (uint);
function seize(address liquidator, address borrower, uint seizeTokens) external returns (uint);

/***
 * Admin Functions
 */

function _setPendingAdmin(address payable newPendingAdmin) external returns (uint);
function _acceptAdmin() external returns (uint);
function _setComptroller(ComptrollerInterface newComptroller) public returns (uint);
function _setReserveFactor(uint newReserveFactorMantissa) external returns (uint);
function _reduceReserves(uint reduceAmount) external returns (uint);
function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint);
```

```

}

contract CErc20Storage {
    /**
     * @notice Underlying asset for this CToken
     */
    address public underlying;
}

contract CErc20Interface is CErc20Storage {
    /** User Interface ***/

    function mint(uint mintAmount) external returns (uint);
    function redeem(uint redeemTokens) external returns (uint);
    function redeemUnderlying(uint redeemAmount) external returns (uint);
    function borrow(uint borrowAmount) external returns (uint);
    function repayBorrow(uint repayAmount) external returns (uint);
    function repayBorrowBehalf(address borrower, uint repayAmount) external returns (uint);
    function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) external
    returns (uint);

    /** Admin Functions ***/

    function _addReserves(uint addAmount) external returns (uint);
}

contract CDelegationStorage {
    /**
     * @notice Implementation address for this contract
     */
    address public implementation;
}

contract CDelegatorInterface is CDelegationStorage {
    /**
     * @notice Emitted when implementation is changed
     */
    event NewImplementation(address oldImplementation, address newImplementation);

    /**
     * @notice Called by the admin to update the implementation of the delegator
     * @param implementation_ The address of the new implementation for delegation
     * @param allowResign Flag to indicate whether to call _resignImplementation on the old implementation
     * @param becomeImplementationData The encoded bytes data to be passed to _becomeImplementation
     */
    function _setImplementation(address implementation_, bool allowResign, bytes memory
becomeImplementationData) public;
}

contract CDelegateInterface is CDelegationStorage {
    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @dev Should revert if any issues arise which make it unfit for delegation
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public;

    /**
     * @notice Called by the delegator on a delegate to forfeit its responsibility
     */
    function _resignImplementation() public;
}

DAIInterestRateModelV2.sol
pragma solidity ^0.5.16;

import "./JumpRateModel.sol";
import "./SafeMath.sol";

/**
 * @title Compound's DAIInterestRateModel Contract (version 2)
 * @author Compound (modified by Dharma Labs)
 * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper.
 * Version 2 modifies the original interest rate model by increasing the "gap" or slope of the model prior
 * to the "kink" from 0.05% to 2% with the goal of "smoothing out" interest rate changes as the utilization
 * rate increases.
 */
contract DAIInterestRateModelV2 is JumpRateModel {
    using SafeMath for uint;

    /**
     * @notice The additional margin per block separating the base borrow rate from the roof (2% / block).
     * Note that this value has been increased from the original value of 0.05% per block.
    */
}

```

```

/*
uint public constant gapPerBlock = 2e16 / blocksPerYear;

/**
 * @notice The assumed (1 - reserve factor) used to calculate the minimum borrow rate (reserve factor = 0.05)
 */
uint public constant assumedOneMinusReserveFactorMantissa = 0.95e18;

PotLike pot;
JugLike jug;

/**
 * @notice Construct an interest rate model
 * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
 * @param kink_ The utilization point at which the jump multiplier is applied
 * @param pot_ The address of the Dai pot (where DSR is earned)
 * @param jug_ The address of the Dai jug (where SF is kept)
 */
constructor(uint jumpMultiplierPerYear, uint kink_, address pot_, address jug_) JumpRateModel(0, 0, jumpMultiplierPerYear, kink_) public {
    pot = PotLike(pot_);
    jug = JugLike(jug_);
    poke();
}

/**
 * @notice Calculates the current supply interest rate per block including the Dai savings rate
 * @param cash The total amount of cash the market has
 * @param borrows The total amount of borrows the market has outstanding
 * @param reserves The total amount of reserves the market has
 * @param reserveFactorMantissa The current reserve factor the market has
 * @return The supply rate per block (as a percentage, and scaled by 1e18)
 */
function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) public view returns (uint) {
    uint protocolRate = super.getSupplyRate(cash, borrows, reserves, reserveFactorMantissa);

    uint underlying = cash.add(borrows).sub(reserves);
    if(underlying == 0) {
        return protocolRate;
    } else {
        uint cashRate = cash.mul(dsrPerBlock()).div(underlying);
        return cashRate.add(protocolRate);
    }
}

/**
 * @notice Calculates the Dai savings rate per block
 * @return The Dai savings rate per block (as a percentage, and scaled by 1e18)
 */
function dsrPerBlock() public view returns (uint) {
    return pot
        .dsr().sub(1e27) //scaled 1e27 aka RAY, and includes an extra "ONE" before subtraction
        .div(1e9) //descale to 1e18
        .mul(15); //15 seconds per block
}

/**
 * @notice Resets the baseRate and multiplier per block based on the stability fee and Dai savings rate
 */
function poke() public {
    (uint duty,) = jug.ilks("ETH-A");
    uint stabilityFeePerBlock = duty.add(jug.base()).sub(1e27).mul(1e18).div(1e27).mul(15);

    // We ensure the minimum borrow rate >= DSR / (1 - reserve factor)
    baseRatePerBlock = dsrPerBlock().mul(1e18).div(assumedOneMinusReserveFactorMantissa);

    // The roof borrow rate is max(base rate, stability fee) + gap, from which we derive the slope
    if(baseRatePerBlock < stabilityFeePerBlock) {
        multiplierPerBlock =
            stabilityFeePerBlock.sub(baseRatePerBlock).add(gapPerBlock).mul(1e18).div(kink);
    } else {
        multiplierPerBlock = gapPerBlock.mul(1e18).div(kink);
    }

    emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);
}
}

/*** Maker Interfaces ***/

contract PotLike {
    function chi() external view returns (uint);
    function dsr() external view returns (uint);
    function rho() external view returns (uint);
}

```

```

function pie(address) external view returns (uint);
function drip() external returns (uint);
function join(uint) external;
function exit(uint) external;
}

contract JugLike {
    // --- Data ---
    struct Ilk {
        uint256 duty;
        uint256 rho;
    }

    mapping (bytes32 => Ilk) public ilks;
    uint256 public base;
}

EIP20Interface.sol
pragma solidity ^0.5.16;

/**
 * @title ERC 20 Token Standard Interface
 * https://eips.ethereum.org/EIPS/eip-20
 */
interface EIP20Interface {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);

    /**
     * @notice Get the total number of tokens in circulation
     * @return The supply of tokens
     */
    function totalSupply() external view returns (uint256);

    /**
     * @notice Gets the balance of the specified address
     * @param owner The address from which the balance will be retrieved
     * @return The balance
     */
    function balanceOf(address owner) external view returns (uint256 balance);

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 amount) external returns (bool success);

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(address src, address dst, uint256 amount) external returns (bool success);

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     * and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param amount The number of tokens that are approved (-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 amount) external returns (bool success);

    /**
     * @notice Get the current allowance from `owner` for `spender`
     * @param owner The address of the account which owns the tokens to be spent
     * @param spender The address of the account which may transfer tokens
     * @return The number of tokens allowed to be spent (-1 means infinite)
     */
    function allowance(address owner, address spender) external view returns (uint256 remaining);

    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);
}

EIP20NonStandardInterface.sol
pragma solidity ^0.5.16;

```

```

/**
 * @title EIP20NonStandardInterface
 * @dev Version of ERC20 with no return values for `transfer` and `transferFrom`.
 * See https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
 */
interface EIP20NonStandardInterface {

    /**
     * @notice Get the total number of tokens in circulation
     * @return The supply of tokens
     */
    function totalSupply() external view returns (uint256);

    /**
     * @notice Gets the balance of the specified address
     * @param owner The address from which the balance will be retrieved
     * @return The balance
     */
    function balanceOf(address owner) external view returns (uint256 balance);

    /**
     * @notice !!! NOTICE !!! `transfer` does not return a value, in violation of the ERC-20 specification
     */

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     */
    function transfer(address dst, uint256 amount) external;

    /**
     * @notice !!! NOTICE !!! `transferFrom` does not return a value, in violation of the ERC-20 specification
     */

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param amount The number of tokens to transfer
     */
    function transferFrom(address src, address dst, uint256 amount) external;

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     * and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param amount The number of tokens that are approved
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 amount) external returns (bool success);

    /**
     * @notice Get the current allowance from `owner` for `spender`
     * @param owner The address of the account which owns the tokens to be spent
     * @param spender The address of the account which may transfer tokens
     * @return The number of tokens allowed to be spent
     */
    function allowance(address owner, address spender) external view returns (uint256 remaining);

    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);
}

```

*ErrorReporter.sol*

```

pragma solidity ^0.5.16;

contract ComptrollerErrorReporter {
    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        COMPTROLLER_MISMATCH,
        INSUFFICIENT_SHORTFALL,
        INSUFFICIENT_LIQUIDITY,
        INVALID_CLOSE_FACTOR,
        INVALID_COLLATERAL_FACTOR,
        INVALID_LIQUIDATION_INCENTIVE,
        MARKET_NOT_ENTERED, // no longer possible
        MARKET_NOT_LISTED,
    }
}

```

```

    MARKET_ALREADY_LISTED,
    MATH_ERROR,
    NONZERO_BORROW_BALANCE,
    PRICE_ERROR,
    REJECTION,
    SNAPSHOT_ERROR,
    TOO_MANY_ASSETS,
    TOO MUCH REPAY
}

enum FailureInfo {
    ACCEPT_ADMIN_PENDING_ADMIN_CHECK,
    ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK,
    EXIT_MARKET_BALANCE_OWED,
    EXIT_MARKET_REJECTION,
    SET_CLOSE_FACTOR_OWNER_CHECK,
    SET_CLOSE_FACTOR_VALIDATION,
    SET_COLLATERAL_FACTOR_OWNER_CHECK,
    SET_COLLATERAL_FACTOR_NO_EXISTS,
    SET_COLLATERAL_FACTOR_VALIDATION,
    SET_COLLATERAL_FACTOR_WITHOUT_PRICE,
    SET_IMPLEMENTATION_OWNER_CHECK,
    SET_LIQUIDATION_INCENTIVE_OWNER_CHECK,
    SET_LIQUIDATION_INCENTIVE_VALIDATION,
    SET_MAX_ASSETS_OWNER_CHECK,
    SET_PENDING_ADMIN_OWNER_CHECK,
    SET_PENDING_IMPLEMENTATION_OWNER_CHECK,
    SET_PRICE_ORACLE_OWNER_CHECK,
    SUPPORT_MARKET_EXISTS,
    SUPPORT_MARKET_OWNER_CHECK,
    SET_PAUSE_GUARDIAN_OWNER_CHECK
}

/**
 * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail` is an
arbitrary
 * contract-specific code that enables us to report opaque error codes from upgradeable contracts.
 */
event Failure(uint error, uint info, uint detail);

/**
 * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
function fail(Error err, FailureInfo info) internal returns (uint) {
    emit Failure(uint(err), uint(info), 0);
    return uint(err);
}

/**
 * @dev use this when reporting an opaque error from an upgradeable collaborator contract
function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
    emit Failure(uint(err), uint(info), opaqueError);
    return uint(err);
}

contract TokenErrorReporter {
    enum Error {
        NO_ERROR,
        UNAUTHORIZED,
        BAD_INPUT,
        COMPTROLLER_REJECTION,
        COMPTROLLER_CALCULATION_ERROR,
        INTEREST_RATE_MODEL_ERROR,
        INVALID_ACCOUNT_PAIR,
        INVALID_CLOSE_AMOUNT_REQUESTED,
        INVALID_COLLATERAL_FACTOR,
        MATH_ERROR,
        MARKET_NOT_FRESH,
        MARKET_NOT_LISTED,
        TOKEN_INSUFFICIENT_ALLOWANCE,
        TOKEN_INSUFFICIENT_BALANCE,
        TOKEN_INSUFFICIENT_CASH,
        TOKEN_TRANSFER_IN_FAILED,
        TOKEN_TRANSFER_OUT_FAILED
    }

    /*
     * Note: FailureInfo (but not Error) is kept in alphabetical order
     * This is because FailureInfo grows significantly faster, and
     * the order of Error has some meaning, while the order of FailureInfo
     * is entirely arbitrary.
     */
    enum FailureInfo {

```

```

ACCEPADMIN PENDING ADMIN CHECK,
ACCRUE INTEREST ACCUMULATED INTEREST CALCULATION FAILED,
ACCRUE INTEREST BORROW RATE CALCULATION FAILED,
ACCRUE INTEREST NEW BORROW INDEX CALCULATION FAILED,
ACCRUE INTEREST NEW TOTAL BORROWS CALCULATION FAILED,
ACCRUE INTEREST NEW TOTAL RESERVES CALCULATION FAILED,
ACCRUE INTEREST SIMPLE INTEREST FACTOR CALCULATION FAILED,
BORROW ACCUMULATED BALANCE CALCULATION FAILED,
BORROW ACCRUE INTEREST FAILED,
BORROW CASH NOT AVAILABLE,
BORROW FRESHNESS CHECK,
BORROW NEW TOTAL BALANCE CALCULATION FAILED,
BORROW NEW ACCOUNT BORROW_BALANCE_CALCULATION_FAILED,
BORROW MARKET NOT LISTED,
BORROW COMPTROLLER REJECTION,
LIQUIDATE ACCRUE BORROW INTEREST FAILED,
LIQUIDATE ACCRUE COLLATERAL INTEREST FAILED,
LIQUIDATE COLLATERAL FRESHNESS CHECK,
LIQUIDATE COMPTROLLER REJECTION,
LIQUIDATE COMPTROLLER_CALCULATE_AMOUNT_SEIZE FAILED,
LIQUIDATE CLOSE AMOUNT IS UINT MAX,
LIQUIDATE CLOSE AMOUNT IS ZERO,
LIQUIDATE FRESHNESS CHECK,
LIQUIDATE LIQUIDATOR_IS_BORROWER,
LIQUIDATE REPAY BORROW FRESH FAILED,
LIQUIDATE SEIZE BALANCE INCREMENT FAILED,
LIQUIDATE SEIZE BALANCE DECREMENT FAILED,
LIQUIDATE SEIZE COMPTROLLER REJECTION,
LIQUIDATE SEIZE LIQUIDATOR_IS_BORROWER,
LIQUIDATE SEIZE TOO MUCH,
MINT ACCRUE INTEREST FAILED,
MINT COMPTROLLER REJECTION,
MINT EXCHANGE CALCULATION FAILED,
MINT EXCHANGE_RATE READ FAILED,
MINT FRESHNESS CHECK,
MINT NEW ACCOUNT BALANCE CALCULATION FAILED,
MINT NEW TOTAL SUPPLY CALCULATION FAILED,
MINT TRANSFER_IN FAILED,
MINT TRANSFER_IN NOT POSSIBLE,
REDEEM ACCRUE INTEREST FAILED,
REDEEM COMPTROLLER REJECTION,
REDEEM EXCHANGE TOKENS CALCULATION FAILED,
REDEEM EXCHANGE_AMOUNT CALCULATION FAILED,
REDEEM EXCHANGE_RATE READ FAILED,
REDEEM FRESHNESS CHECK,
REDEEM NEW ACCOUNT BALANCE CALCULATION FAILED,
REDEEM NEW TOTAL SUPPLY CALCULATION FAILED,
REDEEM TRANSFER_OUT NOT POSSIBLE,
REDUCE RESERVES ACCRUE INTEREST FAILED,
REDUCE RESERVES ADMIN CHECK,
REDUCE RESERVES CASH NOT AVAILABLE,
REDUCE RESERVES FRESH CHECK,
REDUCE RESERVES VALIDATION,
REPAY BEHALF ACCRUE INTEREST FAILED,
REPAY BORROW ACCRUE INTEREST FAILED,
REPAY BORROW ACCUMULATED BALANCE CALCULATION FAILED,
REPAY BORROW COMPTROLLER REJECTION,
REPAY BORROW FRESHNESS CHECK,
REPAY BORROW NEW ACCOUNT BORROW BALANCE CALCULATION FAILED,
REPAY BORROW NEW TOTAL BALANCE CALCULATION FAILED,
REPAY BORROW_TRANSFER_IN NOT POSSIBLE,
SET COLLATERAL_FACTOR OWNER CHECK,
SET COLLATERAL_FACTOR VALIDATION,
SET COMPTROLLER OWNER CHECK,
SET INTEREST RATE MODEL ACCRUE INTEREST FAILED,
SET INTEREST RATE MODEL FRESH CHECK,
SET INTEREST RATE MODEL OWNER CHECK,
SET MAX ASSETS OWNER CHECK,
SET ORACLE MARKET NOT LISTED,
SET PENDING ADMIN OWNER CHECK,
SET RESERVE_FACTOR ACCRUE INTEREST FAILED,
SET RESERVE_FACTOR ADMIN CHECK,
SET RESERVE_FACTOR FRESH CHECK,
SET RESERVE_FACTOR BOUNDS CHECK,
TRANSFER COMPTROLLER REJECTION,
TRANSFER NOT ALLOWED,
TRANSFER NOT ENOUGH,
TRANSFER TOO MUCH,
ADD RESERVES ACCRUE INTEREST FAILED,
ADD RESERVES FRESH CHECK,
ADD RESERVES_TRANSFER_IN NOT POSSIBLE
}

/**
 * @dev `error` corresponds to enum Error; `info` corresponds to enum FailureInfo, and `detail` is an
 * arbitrary
 * contract-specific code that enables us to report opaque error codes from upgradeable contracts.
 */

```

```

    /**
event Failure(uint error, uint info, uint detail);

/**
 * @dev use this when reporting a known error from the money market or a non-upgradeable collaborator
 */
function fail(Error err, FailureInfo info) internal returns (uint) {
    emit Failure(uint(err), uint(info), 0);
    return uint(err);
}

/**
 * @dev use this when reporting an opaque error from an upgradeable collaborator contract
 */
function failOpaque(Error err, FailureInfo info, uint opaqueError) internal returns (uint) {
    emit Failure(uint(err), uint(info), opaqueError);
    return uint(err);
}
}

Exponential.sol
pragma solidity ^0.5.16;
import "./CarefulMath.sol";

/**
 * @title Exponential module for storing fixed-precision decimals
 * @author Compound
 * @notice Exp is a struct which stores decimals with a fixed precision of 18 decimal places.
 *         Thus, if we wanted to store the 5.1, mantissa would store 5.1e18. That is:
 *         `Exp({mantissa: 510000000000000000})`.
 */
contract Exponential is CarefulMath {
    uint constant expScale = 1e18;
    uint constant doubleScale = 1e36;
    uint constant halfExpScale = expScale/2;
    uint constant mantissaOne = expScale;

    struct Exp {
        uint mantissa;
    }

    struct Double {
        uint mantissa;
    }

    /**
     * @dev Creates an exponential from numerator and denominator values.
     * Note: Returns an error if `num` * 10e18) > MAX_INT,
     *       or if `denom` is zero.
     */
    function getExp(uint num, uint denom) pure internal returns (MathError, Exp memory) {
        (MathError err0, uint scaledNumerator) = mulUInt(num, expScale);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }

        (MathError err1, uint rational) = divUInt(scaledNumerator, denom);
        if (err1 != MathError.NO_ERROR) {
            return (err1, Exp({mantissa: 0}));
        }

        return (MathError.NO_ERROR, Exp({mantissa: rational}));
    }

    /**
     * @dev Adds two exponentials, returning a new exponential.
     */
    function addExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
        (MathError error, uint result) = addUInt(a.mantissa, b.mantissa);

        return (error, Exp({mantissa: result}));
    }

    /**
     * @dev Subtracts two exponentials, returning a new exponential.
     */
    function subExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
        (MathError error, uint result) = subUInt(a.mantissa, b.mantissa);

        return (error, Exp({mantissa: result}));
    }
}

```

```


    /**
     * @dev Multiply an Exp by a scalar, returning a new Exp.
     */
    function mulScalar(Exp memory a, uint scalar) pure internal returns (MathError, Exp memory) {
        (MathError err0, uint scaledMantissa) = mulUInt(a.mantissa, scalar);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }
        return (MathError.NO_ERROR, Exp({mantissa: scaledMantissa}));
    }

    /**
     * @dev Multiply an Exp by a scalar, then truncate to return an unsigned integer.
     */
    function mulScalarTruncate(Exp memory a, uint scalar) pure internal returns (MathError, uint) {
        (MathError err, Exp memory product) = mulScalar(a, scalar);
        if (err != MathError.NO_ERROR) {
            return (err, 0);
        }
        return (MathError.NO_ERROR, truncate(product));
    }

    /**
     * @dev Multiply an Exp by a scalar, truncate, then add an to an unsigned integer, returning an unsigned
     * integer.
     */
    function mulScalarTruncateAddUInt(Exp memory a, uint scalar, uint addend) pure internal returns
    (MathError, uint) {
        (MathError err, Exp memory product) = mulScalar(a, scalar);
        if (err != MathError.NO_ERROR) {
            return (err, 0);
        }
        return addUInt(truncate(product), addend);
    }

    /**
     * @dev Divide an Exp by a scalar, returning a new Exp.
     */
    function divScalar(Exp memory a, uint scalar) pure internal returns (MathError, Exp memory) {
        (MathError err0, uint descaledMantissa) = divUInt(a.mantissa, scalar);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }
        return (MathError.NO_ERROR, Exp({mantissa: descaledMantissa}));
    }

    /**
     * @dev Divide a scalar by an Exp, returning a new Exp.
     */
    function divScalarByExp(uint scalar, Exp memory divisor) pure internal returns (MathError, Exp memory) {
        /*
        We are doing this as:
        getExp(mulUInt(expScale, scalar), divisor.mantissa)

        How it works:
        Exp = a / b;
        Scalar = s;
        `s / (a / b)` = `b * s / a` and since for an Exp `a = mantissa, b = expScale`
        */
        (MathError err0, uint numerator) = mulUInt(expScale, scalar);
        if (err0 != MathError.NO_ERROR) {
            return (err0, Exp({mantissa: 0}));
        }
        return getExp(numerator, divisor.mantissa);
    }

    /**
     * @dev Divide a scalar by an Exp, then truncate to return an unsigned integer.
     */
    function divScalarByExpTruncate(uint scalar, Exp memory divisor) pure internal returns (MathError, uint) {
        (MathError err, Exp memory fraction) = divScalarByExp(scalar, divisor);
        if (err != MathError.NO_ERROR) {
            return (err, 0);
        }
        return (MathError.NO_ERROR, truncate(fraction));
    }

    /**
     * @dev Multiplies two exponentials, returning a new exponential.
     */
    function mulExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {


```

```

(MathError err0, uint doubleScaledProduct) = mulUInt(a.mantissa, b.mantissa);
if (err0 != MathError.NO_ERROR) {
    return (err0, Exp({mantissa: 0}));
}

// We add half the scale before dividing so that we get rounding instead of truncation.
// See "Listing 6" and text above it at https://accu.org/index.php/journals/1717
// Without this change, a result like 6.6...e-19 will be truncated to 0 instead of being rounded to 1e-18.
(MathError err1, uint doubleScaledProductWithHalfScale) = addUInt(halfExpScale,
doubleScaledProduct);
if (err1 != MathError.NO_ERROR) {
    return (err1, Exp({mantissa: 0}));
}

(MathError err2, uint product) = divUInt(doubleScaledProductWithHalfScale, expScale);
// The only error `div` can return is MathError.DIVISION_BY_ZERO but we control `expScale` and it is
not zero.
assert(err2 == MathError.NO_ERROR);

return (MathError.NO_ERROR, Exp({mantissa: product}));
}

/**
 * @dev Multiplies two exponentials given their mantissas, returning a new exponential.
 */
function mulExp(uint a, uint b) pure internal returns (MathError, Exp memory) {
    return mulExp(Exp({mantissa: a}), Exp({mantissa: b}));
}

/**
 * @dev Multiplies three exponentials, returning a new exponential.
 */
function mulExp3(Exp memory a, Exp memory b, Exp memory c) pure internal returns (MathError, Exp
memory) {
    (MathError err, Exp memory ab) = mulExp(a, b);
    if (err != MathError.NO_ERROR) {
        return (err, ab);
    }
    return mulExp(ab, c);
}

/**
 * @dev Divides two exponentials, returning a new exponential.
 *      (a/scale) / (b/scale) = (a/scale) * (scale/b) = a/b,
 *      which we can scale as an Exp by calling getExp(a.mantissa, b.mantissa)
 */
function divExp(Exp memory a, Exp memory b) pure internal returns (MathError, Exp memory) {
    return getExp(a.mantissa, b.mantissa);
}

/**
 * @dev Truncates the given exp to a whole number value.
 *      For example, truncate(Exp{mantissa: 15 * expScale}) = 15
 */
function truncate(Exp memory exp) pure internal returns (uint) {
    // Note: We are not using careful math here as we're performing a division that cannot fail
    return exp.mantissa / expScale;
}

/**
 * @dev Checks if first Exp is less than second Exp.
 */
function lessThanExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa < right.mantissa;
}

/**
 * @dev Checks if left Exp <= right Exp.
 */
function lessThanOrEqualExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa <= right.mantissa;
}

/**
 * @dev Checks if left Exp > right Exp.
 */
function greaterThanExp(Exp memory left, Exp memory right) pure internal returns (bool) {
    return left.mantissa > right.mantissa;
}

/**
 * @dev Returns true if Exp is exactly zero
 */
function isZeroExp(Exp memory value) pure internal returns (bool) {
    return value.mantissa == 0;
}

```

```
function safe224(uint n, string memory errorMessage) pure internal returns (uint224) {
    require(n < 2**224, errorMessage);
    return uint224(n);
}

function safe32(uint n, string memory errorMessage) pure internal returns (uint32) {
    require(n < 2**32, errorMessage);
    return uint32(n);
}

function add_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
    return Exp({mantissa: add_(a.mantissa, b.mantissa)}); 
}

function add_(Double memory a, Double memory b) pure internal returns (Double memory) {
    return Double({mantissa: add_(a.mantissa, b.mantissa)}); 
}

function add_(uint a, uint b) pure internal returns (uint) {
    return add_(a, b, "addition overflow");
}

function add_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
    uint c = a + b;
    require(c >= a, errorMessage);
    return c;
}

function sub_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
    return Exp({mantissa: sub_(a.mantissa, b.mantissa)}); 
}

function sub_(Double memory a, Double memory b) pure internal returns (Double memory) {
    return Double({mantissa: sub_(a.mantissa, b.mantissa)}); 
}

function sub_(uint a, uint b) pure internal returns (uint) {
    return sub_(a, b, "subtraction underflow");
}

function sub_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
    require(b <= a, errorMessage);
    return a - b;
}

function mul_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
    return Exp({mantissa: mul_(a.mantissa, b.mantissa) / expScale}); 
}

function mul_(Exp memory a, uint b) pure internal returns (Exp memory) {
    return Exp({mantissa: mul_(a.mantissa, b)}); 
}

function mul_(uint a, Exp memory b) pure internal returns (uint) {
    return mul_(a, b.mantissa) / expScale;
}

function mul_(Double memory a, Double memory b) pure internal returns (Double memory) {
    return Double({mantissa: mul_(a.mantissa, b.mantissa) / doubleScale}); 
}

function mul_(Double memory a, uint b) pure internal returns (Double memory) {
    return Double({mantissa: mul_(a.mantissa, b)}); 
}

function mul_(uint a, Double memory b) pure internal returns (uint) {
    return mul_(a, b.mantissa) / doubleScale;
}

function mul_(uint a, uint b) pure internal returns (uint) {
    return mul_(a, b, "multiplication overflow");
}

function mul_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {
    if(a == 0 || b == 0) {
        return 0;
    }
    uint c = a * b;
    require(c / a == b, errorMessage);
    return c;
}

function div_(Exp memory a, Exp memory b) pure internal returns (Exp memory) {
    return Exp({mantissa: div_(mul_(a.mantissa, expScale), b.mantissa)}); 
}

function div_(Exp memory a, uint b) pure internal returns (Exp memory) {
```

```

        return Exp({mantissa: div_(a.mantissa, b)});  

    }  

    function div_(uint a, Exp memory b) pure internal returns (uint) {  

        return div_(mul_(a, expScale), b.mantissa);  

    }  

    function div_(Double memory a, Double memory b) pure internal returns (Double memory) {  

        return Double({mantissa: div_(mul_(a.mantissa, doubleScale), b.mantissa)});  

    }  

    function div_(Double memory a, uint b) pure internal returns (Double memory) {  

        return Double({mantissa: div_(a.mantissa, b)});  

    }  

    function div_(uint a, Double memory b) pure internal returns (uint) {  

        return div_(mul_(a, doubleScale), b.mantissa);  

    }  

    function div_(uint a, uint b) pure internal returns (uint) {  

        return div_(a, b, "divide by zero");  

    }  

    function div_(uint a, uint b, string memory errorMessage) pure internal returns (uint) {  

        require(b > 0, errorMessage);  

        return a / b;  

    }  

    function fraction(uint a, uint b) pure internal returns (Double memory) {  

        return Double({mantissa: div_(mul_(a, doubleScale), b)});  

    }  

}

```

### InterestRateModel.sol

```

pragma solidity ^0.5.16;  

/**  

 * @title Compound's InterestRateModel Interface  

 * @author Compound  

 */  

contract InterestRateModel {  

    /// @notice Indicator that this is an InterestRateModel contract (for inspection)  

    bool public constant isInterestRateModel = true;  

    /**  

     * @notice Calculates the current borrow interest rate per block  

     * @param cash The total amount of cash the market has  

     * @param borrows The total amount of borrows the market has outstanding  

     * @param reserves The total amount of reserves the market has  

     * @return The borrow rate per block (as a percentage, and scaled by 1e18)  

     */  

    function getBorrowRate(uint cash, uint borrows, uint reserves) external view returns (uint);  

    /**  

     * @notice Calculates the current supply interest rate per block  

     * @param cash The total amount of cash the market has  

     * @param borrows The total amount of borrows the market has outstanding  

     * @param reserves The total amount of reserves the market has  

     * @param reserveFactorMantissa The current reserve factor the market has  

     * @return The supply rate per block (as a percentage, and scaled by 1e18)  

     */  

    function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) external view returns (uint);
}

```

### JumpRateModel.sol

```

pragma solidity ^0.5.16;  

import "./InterestRateModel.sol";  

import "./SafeMath.sol";  

/**  

 * @title Compound's JumpRateModel Contract  

 * @author Compound  

 */  

contract JumpRateModel is InterestRateModel {  

    using SafeMath for uint;  

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock, uint jumpMultiplierPerBlock, uint kink);
}

```

```
* @notice The approximate number of blocks per year that is assumed by the interest rate model
*/
uint public constant blocksPerYear = 2102400;

/**
 * @notice The multiplier of utilization rate that gives the slope of the interest rate
 */
uint public multiplierPerBlock;

/**
 * @notice The base interest rate which is the y-intercept when utilization rate is 0
 */
uint public baseRatePerBlock;

/**
 * @notice The multiplierPerBlock after hitting a specified utilization point
 */
uint public jumpMultiplierPerBlock;

/**
 * @notice The utilization point at which the jump multiplier is applied
 */
uint public kink;

/**
 * @notice Construct an interest rate model
 * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
 * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
 * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
 * @param kink_ The utilization point at which the jump multiplier is applied
 */
constructor(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_) public {
    baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
    multiplierPerBlock = multiplierPerYear.div(blocksPerYear);
    jumpMultiplierPerBlock = jumpMultiplierPerYear.div(blocksPerYear);
    kink = kink_;
}

emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock, kink);

/**
 * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market (currently unused)
 * @return The utilization rate as a mantissa between [0, 1e18]
 */
function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
    // Utilization rate is 0 when there are no borrows
    if (borrows == 0) {
        return 0;
    }
    return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
}

/**
 * @notice Calculates the current borrow rate per block, with the error code expected by the market
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market
 * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
 */
function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
    uint util = utilizationRate(cash, borrows, reserves);

    if (util <= kink) {
        return util.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
    } else {
        uint normalRate = kink.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
        uint excessUtil = util.sub(kink);
        return excessUtil.mul(jumpMultiplierPerBlock).div(1e18).add(normalRate);
    }
}

/**
 * @notice Calculates the current supply rate per block
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market
 * @param reserveFactorMantissa The current reserve factor for the market
 * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
 */
function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) public view
returns (uint) {
    uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
    uint borrowRate = getBorrowRate(cash, borrows, reserves);
```

```

        uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
        return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
    }

}

Maximillion.sol
pragma solidity ^0.5.16;
import "./CEther.sol";
<**
 * @title Compound's Maximillion Contract
 * @author Compound
 */
contract Maximillion {
<**
 * @notice The default cEther market to repay in
 */
CEther public cEther;

<**
 * @notice Construct a Maximillion to repay max in a CEther market
 */
constructor(CEther cEther_) public {
    cEther = cEther_;
}

<**
 * @notice msg.sender sends Ether to repay an account's borrow in the cEther market
 * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
 * @param borrower The address of the borrower account to repay on behalf of
 */
function repayBehalf(address borrower) public payable {
    repayBehalfExplicit(borrower, cEther);
}

<**
 * @notice msg.sender sends Ether to repay an account's borrow in a cEther market
 * @dev The provided Ether is applied towards the borrow balance, any excess is refunded
 * @param borrower The address of the borrower account to repay on behalf of
 * @param cEther_ The address of the cEther contract to repay in
 */
function repayBehalfExplicit(address borrower, CEther cEther_) public payable {
    uint received = msg.value;
    uint borrows = cEther_.borrowBalanceCurrent(borrower);
    if(received > borrows) {
        cEther_.repayBorrowBehalf.value(borrows)(borrower);
        msg.sender.transfer(received - borrows);
    } else {
        cEther_.repayBorrowBehalf.value(received)(borrower);
    }
}

Migrations.sol
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.8.0;

contract Migrations {
    address public owner = msg.sender;
    uint public last_completed_migration;

    modifier restricted() {
        require(
            msg.sender == owner,
            "This function is restricted to the contract's owner"
        );
    }

    function setCompleted(uint completed) public restricted {
        last_completed_migration = completed;
    }
}

PriceOracle.sol
pragma solidity ^0.5.16;
import "./CToken.sol";
contract PriceOracle {

```

```

///@notice Indicator that this is a PriceOracle contract (for inspection)
bool public constant isPriceOracle = true;

/**
 * @notice Get the underlying price of a cToken asset
 * @param cToken The cToken to get the underlying price of
 * @return The underlying asset price mantissa (scaled by 1e18).
 *         Zero means the price is unavailable.
 */
function getUnderlyingPrice(CToken cToken) external view returns (uint);

}

PriceOracleProxy.sol
pragma solidity ^0.5.16;

import "./CErc20.sol";
import "./CToken.sol";
import "./PriceOracle.sol";

interface V1PriceOracleInterface {
    function assetPrices(address asset) external view returns (uint);
}

contract PriceOracleProxy is PriceOracle {
    ///@notice Indicator that this is a PriceOracle contract (for inspection)
    bool public constant isPriceOracle = true;

    ///@notice The v1 price oracle, which will continue to serve prices for v1 assets
    V1PriceOracleInterface public v1PriceOracle;

    ///@notice Address of the guardian, which may set the SAI price once
    address public guardian;

    ///@notice Address of the cEther contract, which has a constant price
    address public cEthAddress;

    ///@notice Address of the cUSDC contract, which we hand pick a key for
    address public cUsdcAddress;

    ///@notice Address of the cUSDT contract, which uses the cUSDC price
    address public cUsdtAddress;

    ///@notice Address of the cSAI contract, which may have its price set
    address public cSaiAddress;

    ///@notice Address of the cDAI contract, which we hand pick a key for
    address public cDaiAddress;

    ///@notice Handpicked key for USDC
    address public constant usdcOracleKey = address(1);

    ///@notice Handpicked key for DAI
    address public constant daiOracleKey = address(2);

    ///@notice Frozen SAI price (or 0 if not set yet)
    uint public saiPrice;

    /**
     * @param guardian_ The address of the guardian, which may set the SAI price once
     * @param v1PriceOracle_ The address of the v1 price oracle, which will continue to operate and hold
     * prices for collateral assets
     * @param cEthAddress_ The address of cETH, which will return a constant 1e18, since all prices relative to
     * ether
     * @param cUsdcAddress_ The address of cUSDC, which will be read from a special oracle key
     * @param cSaiAddress_ The address of cSAI, which may be read directly from storage
     * @param cDaiAddress_ The address of cDAI, which will be read from a special oracle key
     * @param cUsdtAddress_ The address of cUSDT, which uses the cUSDC price
     */
constructor(address guardian_,
            address v1PriceOracle_,
            address cEthAddress_,
            address cUsdcAddress_,
            address cSaiAddress_,
            address cDaiAddress_,
            address cUsdtAddress_) public {
    guardian = guardian_;
    v1PriceOracle = V1PriceOracleInterface(v1PriceOracle_);
    cEthAddress = cEthAddress_;
    cUsdcAddress = cUsdcAddress_;
    cSaiAddress = cSaiAddress_;
    cDaiAddress = cDaiAddress_;
    cUsdtAddress = cUsdtAddress_;
}

```

```


    /**
     * @notice Get the underlying price of a listed cToken asset
     * @param cToken The cToken to get the underlying price of
     * @return The underlying asset price mantissa (scaled by 1e18)
     */
    function getUnderlyingPrice(CToken cToken) public view returns (uint) {
        address cTokenAddress = address(cToken);

        if (cTokenAddress == cEthAddress) {
            // ether always worth 1
            return 1e18;
        }

        if (cTokenAddress == cUsdcAddress || cTokenAddress == cUsdtAddress) {
            return v1PriceOracle.assetPrices(usdcOracleKey);
        }

        if (cTokenAddress == cDaiAddress) {
            return v1PriceOracle.assetPrices(daiOracleKey);
        }

        if (cTokenAddress == cSaiAddress) {
            // use the frozen SAI price if set, otherwise use the DAI price
            return saiPrice > 0 ? saiPrice : v1PriceOracle.assetPrices(daiOracleKey);
        }

        // otherwise just read from v1 oracle
        address underlying = CErc20(cTokenAddress).underlying();
        return v1PriceOracle.assetPrices(underlying);
    }

    /**
     * @notice Set the price of SAI, permanently
     * @param price The price for SAI
     */
    function setSaiPrice(uint price) public {
        require(msg.sender == guardian, "only guardian may set the SAI price");
        require(saiPrice == 0, "SAI price may only be set once");
        require(price < 0.1e18, "SAI price must be < 0.1 ETH");
        saiPrice = price;
    }
}


```

### Reservoir.sol

```


pragma solidity ^0.5.16;

/**
 * @title Reservoir Contract
 * @notice Distributes a token to a different contract at a fixed rate.
 * @dev This contract must be poked via the `drip()` function every so often.
 * @author Compound
 */
contract Reservoir {

    ///@notice The block number when the Reservoir started (immutable)
    uint public dripStart;

    ///@notice Tokens per block that to drip to target (immutable)
    uint public dripRate;

    ///@notice Reference to token to drip (immutable)
    EIP20Interface public token;

    ///@notice Target to receive dripped tokens (immutable)
    address public target;

    ///@notice Amount that has already been dripped
    uint public dripped;

    /**
     * @notice Constructs a Reservoir
     * @param dripRate Numer of tokens per block to drip
     * @param token_ The token to drip
     * @param target_ The recipient of dripped tokens
     */
    constructor(uint dripRate_, EIP20Interface token_, address target_) public {
        dripStart = block.number;
        dripRate = dripRate_;
        token = token_;
        target = target_;
        dripped = 0;
    }

    /**
     * @notice Drips the maximum amount of tokens to match the drip rate since inception
     */


```

```

/*
 * @dev Note: this will only drip up to the amount of tokens available.
 * @return The amount of tokens dripped in this call
 */
function drip() public returns (uint) {
    // First, read storage into memory
    EIP20Interface token_ = token;
    uint reservoirBalance_ = token_.balanceOf(address(this)); // TODO: Verify this is a static call
    uint dripRate_ = dripRate;
    uint dripStart_ = dripStart;
    uint dripped_ = dripped;
    address target_ = target;
    uint blockNumber_ = block.number;

    // Next, calculate intermediate values
    uint dripTotal_ = mul(dripRate_, blockNumber_ - dripStart_, "dripTotal overflow");
    uint deltaDrip_ = sub(dripTotal_, dripped_, "deltaDrip underflow");
    uint toDrip_ = min(reservoirBalance_, deltaDrip_);
    uint drippedNext_ = add(dripped_, toDrip_, "tautological");

    // Finally, write new `dripped` value and transfer tokens to target
    dripped_ = drippedNext_;
    token_.transfer(target_, toDrip_);

    return toDrip_;
}

/* Internal helper functions for safe math */

function add(uint a, uint b, string memory errorMessage) internal pure returns (uint) {
    uint c = a + b;
    require(c >= a, errorMessage);
    return c;
}

function sub(uint a, uint b, string memory errorMessage) internal pure returns (uint) {
    require(b <= a, errorMessage);
    uint c = a - b;
    return c;
}

function mul(uint a, uint b, string memory errorMessage) internal pure returns (uint) {
    if (a == 0) {
        return 0;
    }
    uint c = a * b;
    require(c / a == b, errorMessage);
    return c;
}

function min(uint a, uint b) internal pure returns (uint) {
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}

import "./EIP20Interface.sol";

SafeMath.sol
pragma solidity ^0.5.16;

// From https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/Math.sol
// Subject to the MIT license.

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on overflow.
     *
     * Counterpart to Solidity's `+` operator.
     */
}

```

```
* Requirements:  
* - Addition cannot overflow.  
*/  
function add(uint256 a, uint256 b) internal pure returns (uint256) {  
    uint256 c = a + b;  
    require(c >= a, "SafeMath: addition overflow");  
  
    return c;  
}  
  
/**  
 * @dev Returns the addition of two unsigned integers, reverting with custom message on overflow.  
 * Counterpart to Solidity's `+` operator.  
 * Requirements:  
 * - Addition cannot overflow.  
 */  
function add(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {  
    uint256 c = a + b;  
    require(c >= a, errorMessage);  
  
    return c;  
}  
  
/**  
 * @dev Returns the subtraction of two unsigned integers, reverting on underflow (when the result is negative).  
 * Counterpart to Solidity's `-` operator.  
 * Requirements:  
 * - Subtraction cannot underflow.  
 */  
function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
    return sub(a, b, "SafeMath: subtraction underflow");  
}  
  
/**  
 * @dev Returns the subtraction of two unsigned integers, reverting with custom message on underflow (when the result is negative).  
 * Counterpart to Solidity's `-` operator.  
 * Requirements:  
 * - Subtraction cannot underflow.  
 */  
function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {  
    require(b <= a, errorMessage);  
    uint256 c = a - b;  
  
    return c;  
}  
  
/**  
 * @dev Returns the multiplication of two unsigned integers, reverting on overflow.  
 * Counterpart to Solidity's `*` operator.  
 * Requirements:  
 * - Multiplication cannot overflow.  
 */  
function mul(uint256 a, uint256 b) internal pure returns (uint256) {  
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the  
    // benefit is lost if 'b' is also tested.  
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522  
    if(a == 0) {  
        return 0;  
    }  
  
    uint256 c = a * b;  
    require(c / a == b, "SafeMath: multiplication overflow");  
  
    return c;  
}  
  
/**  
 * @dev Returns the multiplication of two unsigned integers, reverting on overflow.  
 * Counterpart to Solidity's `*` operator.  
 * Requirements:  
 * - Multiplication cannot overflow.  
 */  
function mul(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {  
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the  
    // benefit is lost if 'b' is also tested.
```

```

// See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
if (a == 0) {
    return 0;
}

uint256 c = a * b;
require(c / a == b, errorMessage);

return c;
}

/**
* @dev Returns the integer division of two unsigned integers.
* Reverts on division by zero. The result is rounded towards zero.
*
* Counterpart to Solidity's `/` operator. Note: this function uses a
* `revert` opcode (which leaves remaining gas untouched) while Solidity
* uses an invalid opcode to revert (consuming all remaining gas).
*
* Requirements:
* - The divisor cannot be zero.
*/
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    return div(a, b, "SafeMath: division by zero");
}

/**
* @dev Returns the integer division of two unsigned integers.
* Reverts with custom message on division by zero. The result is rounded towards zero.
*
* Counterpart to Solidity's `/` operator. Note: this function uses a
* `revert` opcode (which leaves remaining gas untouched) while Solidity
* uses an invalid opcode to revert (consuming all remaining gas).
*
* Requirements:
* - The divisor cannot be zero.
*/
function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, errorMessage);
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold

    return c;
}

/**
* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
* Reverts when dividing by zero.
*
* Counterpart to Solidity's `%` operator. This function uses a `revert`
* opcode (which leaves remaining gas untouched) while Solidity uses an
* invalid opcode to revert (consuming all remaining gas).
*
* Requirements:
* - The divisor cannot be zero.
*/
function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    return mod(a, b, "SafeMath: modulo by zero");
}

/**
* @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
* Reverts with custom message when dividing by zero.
*
* Counterpart to Solidity's `%` operator. This function uses a `revert`
* opcode (which leaves remaining gas untouched) while Solidity uses an
* invalid opcode to revert (consuming all remaining gas).
*
* Requirements:
* - The divisor cannot be zero.
*/
function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
    require(b != 0, errorMessage);
    return a % b;
}

SimplePriceOracle.sol
pragma solidity ^0.5.16;

import "./PriceOracle.sol";
import "./CErc20.sol";

contract SimplePriceOracle is PriceOracle {

```

```

mapping(address => uint) prices;
event PricePosted(address asset, uint previousPriceMantissa, uint requestedPriceMantissa, uint newPriceMantissa);

function getUnderlyingPrice(CToken cToken) public view returns (uint) {
    if (compareStrings(cToken.symbol(), "cETH")) {
        return 1e18;
    } else {
        return prices[address(CErc20(address(cToken)).underlying())];
    }
}

function setUnderlyingPrice(CToken cToken, uint underlyingPriceMantissa) public {
    address asset = address(CErc20(address(cToken)).underlying());
    emit PricePosted(asset, prices[asset], underlyingPriceMantissa, underlyingPriceMantissa);
    prices[asset] = underlyingPriceMantissa;
}

function setDirectPrice(address asset, uint price) public {
    emit PricePosted(asset, prices[asset], price, price);
    prices[asset] = price;
}

// v1 price oracle interface for use as backing of proxy
function assetPrices(address asset) external view returns (uint) {
    return prices[asset];
}

function compareStrings(string memory a, string memory b) internal pure returns (bool) {
    return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))));
}
}

```

### Timelock.sol

```

pragma solidity ^0.5.16;
import "./SafeMath.sol";

contract Timelock {
    using SafeMath for uint;

    event NewAdmin(address indexed newAdmin);
    event NewPendingAdmin(address indexed newPendingAdmin);
    event NewDelay(uint indexed newDelay);
    event CancelTransaction(bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uint eta);
    event ExecuteTransaction(bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uint eta);
    event QueueTransaction(bytes32 indexed txHash, address indexed target, uint value, string signature, bytes data, uint eta);

    uint public constant GRACE_PERIOD = 14 days;
    uint public constant MINIMUM_DELAY = 2 days;
    uint public constant MAXIMUM_DELAY = 30 days;

    address public admin;
    address public pendingAdmin;
    uint public delay;

    mapping (bytes32 => bool) public queuedTransactions;

    constructor(address admin_, uint delay_) public {
        require(delay_ >= MINIMUM_DELAY, "Timelock::constructor: Delay must exceed minimum delay.");
        require(delay_ <= MAXIMUM_DELAY, "Timelock::setDelay: Delay must not exceed maximum delay.");

        admin = admin_;
        delay = delay_;
    }

    function() external payable {}

    function setDelay(uint delay_) public {
        require(msg.sender == address(this), "Timelock::setDelay: Call must come from Timelock.");
        require(delay_ >= MINIMUM_DELAY, "Timelock::setDelay: Delay must exceed minimum delay.");
        require(delay_ <= MAXIMUM_DELAY, "Timelock::setDelay: Delay must not exceed maximum delay.");
        delay = delay_;
        emit NewDelay(delay);
    }

    function acceptAdmin() public {
        require(msg.sender == pendingAdmin, "Timelock::acceptAdmin: Call must come from pendingAdmin.");
        admin = msg.sender;
    }
}

```

```

pendingAdmin = address(0);
emit NewAdmin(admin);
}

function setPendingAdmin(address pendingAdmin) public {
    require(msg.sender == address(this), "Timelock::setPendingAdmin: Call must come from Timelock.");
    pendingAdmin = pendingAdmin_;
    emit NewPendingAdmin(pendingAdmin);
}

function queueTransaction(address target, uint value, string memory signature, bytes memory data, uint eta)
public payable returns (bytes32) {
    require(msg.sender == admin, "Timelock::queueTransaction: Call must come from admin.");
    require(eta >= getBlockTimestamp().add(delay), "Timelock::queueTransaction: Estimated execution block must satisfy delay.");
    bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
    queuedTransactions[txHash] = true;
    emit QueueTransaction(txHash, target, value, signature, data, eta);
    return txHash;
}

function cancelTransaction(address target, uint value, string memory signature, bytes memory data, uint eta)
public {
    require(msg.sender == admin, "Timelock::cancelTransaction: Call must come from admin.");
    bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
    queuedTransactions[txHash] = false;
    emit CancelTransaction(txHash, target, value, signature, data, eta);
}

function executeTransaction(address target, uint value, string memory signature, bytes memory data, uint eta)
public payable returns (bytes memory) {
    require(msg.sender == admin, "Timelock::executeTransaction: Call must come from admin.");
    bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
    require(queuedTransactions[txHash], "Timelock::executeTransaction: Transaction hasn't been queued.");
    require(getBlockTimestamp() >= eta, "Timelock::executeTransaction: Transaction hasn't surpassed time lock.");
    require(getBlockTimestamp() <= eta.add(GRACE_PERIOD), "Timelock::executeTransaction: Transaction is stale.");
    queuedTransactions[txHash] = false;
    bytes memory callData;
    if (bytes(signature).length == 0) {
        callData = data;
    } else {
        callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
    }
    // solium-disable-next-line security/no-call-value
    (bool success, bytes memory returnData) = target.call.value(value)(callData);
    require(success, "Timelock::executeTransaction: Transaction execution reverted.");
    emit ExecuteTransaction(txHash, target, value, signature, data, eta);
    return returnData;
}

function getBlockTimestamp() internal view returns (uint) {
    // solium-disable-next-line security/no-block-members
    return block.timestamp;
}
}

Unitroller.sol

pragma solidity ^0.5.16;

import "./ErrorReporter.sol";
import "./ComptrollerStorage.sol";
/***
 * @title ComptrollerCore
 * @dev Storage for the comptroller is at this address, while execution is delegated to the `comptrollerImplementation`.
 * CTokens should reference this contract as their comptroller.
 */
contract Unitroller is UnitrollerAdminStorage, ComptrollerErrorReporter {
    /**

```

```

    * @notice Emitted when pendingComptrollerImplementation is changed
    */
event NewPendingImplementation(address oldPendingImplementation, address newPendingImplementation);

    /**
     * @notice Emitted when pendingComptrollerImplementation is accepted, which means comptroller
     * implementation is updated
     */
event NewImplementation(address oldImplementation, address newImplementation);

    /**
     * @notice Emitted when pendingAdmin is changed
     */
event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

    /**
     * @notice Emitted when pendingAdmin is accepted, which means admin is updated
     */
event NewAdmin(address oldAdmin, address newAdmin);

constructor() public {
    // Set admin to caller
    admin = msg.sender;
}

/** Admin Functions */
function _setPendingImplementation(address newPendingImplementation) public returns (uint) {
    if (msg.sender != admin) {
        return
    }
FailureInfo.SET_PENDING_IMPLEMENTATION_OWNER_CHECK;
}                                     fail(Error.UNAUTHORIZED,)

address oldPendingImplementation = pendingComptrollerImplementation;
pendingComptrollerImplementation = newPendingImplementation;
emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);
return uint(Error.NO_ERROR);
}

    /**
     * @notice Accepts new implementation of comptroller. msg.sender must be pendingImplementation
     * @dev Admin function for new implementation to accept its role as implementation
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
function _acceptImplementation() public returns (uint) {
    // Check caller is pendingImplementation and pendingImplementation != address(0)
    if (msg.sender != pendingComptrollerImplementation || pendingComptrollerImplementation == address(0)) {
        return
    }
FailureInfo.ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK;
}                                     fail(Error.UNAUTHORIZED,)

// Save current values for inclusion in log
address oldImplementation = comptrollerImplementation;
address oldPendingImplementation = pendingComptrollerImplementation;
comptrollerImplementation = pendingComptrollerImplementation;
pendingComptrollerImplementation = address(0);

emit NewImplementation(oldImplementation, comptrollerImplementation);
emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);
return uint(Error.NO_ERROR);
}

    /**
     * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the
     * transfer.
     * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to
     * finalize the transfer.
     * @param newPendingAdmin New pending admin.
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
function _setPendingAdmin(address newPendingAdmin) public returns (uint) {
    // Check caller = admin
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
    }

    // Save current value, if any, for inclusion in log
    address oldPendingAdmin = pendingAdmin;
}

```

```

    // Store pendingAdmin with value newPendingAdmin
    pendingAdmin = newPendingAdmin;

    // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
    emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

    return uint(Error.NO_ERROR);
}

/**
 * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
 * @dev Admin function for pending admin to accept role and update admin
 * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
 */
function _acceptAdmin() public returns (uint) {
    // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
    if (msg.sender != pendingAdmin || msg.sender == address(0)) {
        return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
    }

    // Save current values for inclusion in log
    address oldAdmin = admin;
    address oldPendingAdmin = pendingAdmin;

    // Store admin with value pendingAdmin
    admin = pendingAdmin;

    // Clear the pending value
    pendingAdmin = address(0);

    emit NewAdmin(oldAdmin, admin);
    emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

    return uint(Error.NO_ERROR);
}

/**
 * @dev Delegates execution to an implementation contract.
 * It returns to the external caller whatever the implementation returns
 * or forwards reverts.
 */
function () payable external {
    // delegate all other functions to current implementation
    (bool success,) = comptrollerImplementation.delegatecall(msg.data);

    assembly {
        let free_mem_ptr := mload(0x40)
        returndatocopy(free_mem_ptr, 0, returndatasize)

        switch success
        case 0 { revert(free_mem_ptr, returndatasize) }
        default { return(free_mem_ptr, returndatasize) }
    }
}

```

*WhitePaperInterestRateModel.sol*

```

pragma solidity ^0.5.16;

import "./InterestRateModel.sol";
import "./SafeMath.sol";

/**
 * @title Compound's WhitePaperInterestRateModel Contract
 * @author Compound
 * @notice The parameterized model described in section 2.4 of the original Compound Protocol whitepaper
 */
contract WhitePaperInterestRateModel is InterestRateModel {
    using SafeMath for uint;

    event NewInterestParams(uint baseRatePerBlock, uint multiplierPerBlock);

    /**
     * @notice The approximate number of blocks per year that is assumed by the interest rate model
     */
    uint public constant blocksPerYear = 2102400;

    /**
     * @notice The multiplier of utilization rate that gives the slope of the interest rate
     */
    uint public multiplierPerBlock;

    /**
     * @notice The base interest rate which is the y-intercept when utilization rate is 0
     */
}

```

```
uint public baseRatePerBlock;

/**
 * @notice Construct an interest rate model
 * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
 * @param multiplierPerYear The rate of increase in interest rate wrt utilization (scaled by 1e18)
 */
constructor(uint baseRatePerYear, uint multiplierPerYear) public {
    baseRatePerBlock = baseRatePerYear.div(blocksPerYear);
    multiplierPerBlock = multiplierPerYear.div(blocksPerYear);

    emit NewInterestParams(baseRatePerBlock, multiplierPerBlock);
}

/**
 * @notice Calculates the utilization rate of the market: `borrows / (cash + borrows - reserves)`
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market (currently unused)
 * @return The utilization rate as a mantissa between [0, 1e18]
 */
function utilizationRate(uint cash, uint borrows, uint reserves) public pure returns (uint) {
    // Utilization rate is 0 when there are no borrows
    if(borrows == 0) {
        return 0;
    }

    return borrows.mul(1e18).div(cash.add(borrows).sub(reserves));
}

/**
 * @notice Calculates the current borrow rate per block, with the error code expected by the market
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market
 * @return The borrow rate percentage per block as a mantissa (scaled by 1e18)
 */
function getBorrowRate(uint cash, uint borrows, uint reserves) public view returns (uint) {
    uint ur = utilizationRate(cash, borrows, reserves);
    return ur.mul(multiplierPerBlock).div(1e18).add(baseRatePerBlock);
}

/**
 * @notice Calculates the current supply rate per block
 * @param cash The amount of cash in the market
 * @param borrows The amount of borrows in the market
 * @param reserves The amount of reserves in the market
 * @param reserveFactorMantissa The current reserve factor for the market
 * @return The supply rate percentage per block as a mantissa (scaled by 1e18)
 */
function getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa) public view
returns (uint) {
    uint oneMinusReserveFactor = uint(1e18).sub(reserveFactorMantissa);
    uint borrowRate = getBorrowRate(cash, borrows, reserves);
    uint rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18);
    return utilizationRate(cash, borrows, reserves).mul(rateToPool).div(1e18);
}
```

## 6. 附录 B：安全风险评级标准

| 智能合约漏洞评级标准 |                                                                                                                                                                                                                        |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 漏洞评级       | 漏洞评级说明                                                                                                                                                                                                                 |
| 高危漏洞       | <p>能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失 ETH 或代币的重入漏洞等；</p> <p>能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等；</p> <p>能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送 ETH 导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。</p> |
| 中危漏洞       | 需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等。                                                                                                                                                 |
| 低危漏洞       | 难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量 ETH 或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 gas 触发的事物顺序依赖风险等。                                                                                                                              |

## 7. 附录 C：智能合约安全审计工具简介

### 7.1. Manticore

Manticore 是一个分析二进制文件和智能合约的符号执行工具，Manticore 包含一个符号以太坊虚拟机（EVM），一个 EVM 反汇编器/汇编器以及一个用于自动编译和分析 Solidity 的方便界面。它还集成了 Ethersplay，用于 EVM 字节码的 Bit of Traits of Bits 可视化反汇编程序，用于可视化分析。与二进制文件一样，Manticore 提供了一个简单的命令行界面和一个用于分析 EVM 字节码的 Python API。

### 7.2. Oyente

Oyente 是一个智能合约分析工具，Oyente 可以用来检测智能合约中常见的 bug，比如 reentrancy、事务排序依赖等等。更方便的是，Oyente 的设计是模块化的，所以这让高级用户可以实现并插入他们自己的检测逻辑，以检查他们的合约中自定义的属性。

### 7.3. security.sh

Security 可以验证以太坊智能合约常见的安全问题，例如交易乱序和缺少输入验证，它在全自动化的同时分析程序所有可能的执行路径，此外，Security 还具有用于指定漏洞的特定语言，这使 Security 能够随时关注当前的安全性和其他可靠性问题。

### 7.4. Echidna

Echidna 是一个为了对 EVM 代码进行模糊测试而设计的 Haskell 库。

### 7.5. MAIAN

MAIAN 是一个用于查找以太坊智能合约漏洞的自动化工具，Maian 处理合约的字节码，并尝试建立一系列交易以找出并确认错误。

## 7.6. ethersplay

ethersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

## 7.7. ida-evm

ida-evm 是一个针对以太坊虚拟机 (EVM) 的 IDA 处理器模块。

## 7.8. Remix-ide

Remix 是一款基于浏览器的编译器和 IDE，可让用户使用 Solidity 语言构建以太坊合约并调试交易。

## 7.9. 知道创宇区块链安全审计人员专用工具包

知道创宇渗透测试人员专用工具包，由知道创宇渗透测试工程师研发，收集和使用，包含专用于测试人员的批量自动测试工具，自主研发的工具、脚本或利用工具等。



北京知道创宇信息技术股份有限公司

咨询电话 +86(10)400 060 9587

邮 箱 sec@knownsec.com

官 网 www.knownsec.com

地 址 北京市朝阳区望京 SOHO T2-B座-2509