

```
1 pragma solidity ^0.5.16;
2
3 import "./CToken.sol";
4 import "./ErrorReporter.sol";
5 import "./Exponential.sol";
6 import "./PriceOracle.sol";
7 import "./ComptrollerInterface.sol";
8 import "./ComptrollerStorage.sol";
9 import "./Unitroller.sol";
10 import "./Governance/Comp.sol";
11
12 /**
13  * @title Compound's Comptroller Contract
14  * @author Compound
15  */
16 contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
17     /// @notice Emitted when an admin supports a market
18     event MarketListed(CToken cToken);
19
20     /// @notice Emitted when an account enters a market
21     event MarketEntered(CToken cToken, address account);
22
23     /// @notice Emitted when an account exits a market
24     event MarketExited(CToken cToken, address account);
25
26     /// @notice Emitted when close factor is changed by admin
27     event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);
28
29     /// @notice Emitted when a collateral factor is changed by admin
30     event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFactorMantissa);
31
32     /// @notice Emitted when liquidation incentive is changed by admin
33     event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveMantissa);
34
35     /// @notice Emitted when maxAssets is changed by admin
36     event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);
37
38     /// @notice Emitted when price oracle is changed
39     event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);
40
41     /// @notice Emitted when pause guardian is changed
```

```
1 pragma solidity ^0.5.16;
2
3 import "./CToken.sol";
4 import "./ErrorReporter.sol";
5 import "./Exponential.sol";
6 import "./PriceOracle.sol";
7 import "./ComptrollerInterface.sol";
8 import "./ComptrollerStorage.sol";
9 import "./Unitroller.sol";
10 import "./Governance/Comp.sol";
11
12 /**
13  * @title Compound's Comptroller Contract
14  * @author Compound
15  */
16 contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {
17     /// @notice Emitted when an admin supports a market
18     event MarketListed(CToken cToken);
19
20     /// @notice Emitted when an account enters a market
21     event MarketEntered(CToken cToken, address account);
22
23     /// @notice Emitted when an account exits a market
24     event MarketExited(CToken cToken, address account);
25
26     /// @notice Emitted when close factor is changed by admin
27     event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);
28
29     /// @notice Emitted when a collateral factor is changed by admin
30     event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint newCollateralFactorMantissa);
31
32     /// @notice Emitted when liquidation incentive is changed by admin
33     event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint newLiquidationIncentiveMantissa);
34
35     /// @notice Emitted when maxAssets is changed by admin
36     event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);
37
38     /// @notice Emitted when price oracle is changed
39     event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);
40
41     /// @notice Emitted when pause guardian is changed
```

```

42     event NewPauseGuardian(address oldPauseGuardia
n, address newPauseGuardian);
43
44     /// @notice Emitted when an action is paused g
lobally
45     event ActionPaused(string action, bool pauseSt
ate);
46
47     /// @notice Emitted when an action is paused o
n a market
48     event ActionPaused(CToken cToken, string actio
n, bool pauseState);
49
50     /// @notice Emitted when market comped status
is changed
51     event MarketComped(CToken cToken, bool isCompe
d);
52
53     /// @notice Emitted when COMP rate is changed
54     event NewCompRate(uint oldCompRate, uint newCo
mpRate);
55
56     /// @notice Emitted when a new COMP speed is c
alculated for a market
57     event CompSpeedUpdated(CToken indexed cToken,
uint newSpeed);
58
59     /// @notice Emitted when COMP is distributed t
o a supplier
60     event DistributedSupplierComp(CToken indexed c
Token, address indexed supplier, uint compDelta, u
int compSupplyIndex);
61
62     /// @notice Emitted when COMP is distributed t
o a borrower
63     event DistributedBorrowerComp(CToken indexed c
Token, address indexed borrower, uint compDelta, u
int compBorrowIndex);
64
65     /// @notice The threshold above which the flyw
heel transfers COMP, in wei
66     uint public constant compClaimThreshold = 0.00
1e18;
67
68     /// @notice The initial COMP index for a marke
t
69     uint224 public constant compInitialIndex = 1e3
6;
70
71     // closeFactorMantissa must be strictly greate
r than this value
72     uint internal constant closeFactorMinMantissa
= 0.05e18; // 0.05
73
74     // closeFactorMantissa must not exceed this va
lue
75     uint internal constant closeFactorMaxMantissa
= 0.9e18; // 0.9
76
77     // No collateralFactorMantissa may exceed this
value
78     uint internal constant collateralFactorMaxMant
issa = 0.9e18; // 0.9
79
80     // liquidationIncentiveMantissa must be no les
s than this value
81     uint internal constant liquidationIncentiveMin
Mantissa = 1.0e18; // 1.0

```

```

42     event NewPauseGuardian(address oldPauseGuardia
n, address newPauseGuardian);
43
44     /// @notice Emitted when an action is paused g
lobally
45     event ActionPaused(string action, bool pauseSt
ate);
46
47     /// @notice Emitted when an action is paused o
n a market
48     event ActionPaused(CToken cToken, string actio
n, bool pauseState);
49
50     /// @notice Emitted when market comped status
is changed
51     event MarketComped(CToken cToken, bool isCompe
d);
52
53     /// @notice Emitted when COMP rate is changed
54     event NewCompRate(uint oldCompRate, uint newCo
mpRate);
55
56     /// @notice Emitted when a new COMP speed is c
alculated for a market
57     event CompSpeedUpdated(CToken indexed cToken,
uint newSpeed);
58
59     /// @notice Emitted when COMP is distributed t
o a supplier
60     event DistributedSupplierComp(CToken indexed c
Token, address indexed supplier, uint compDelta, u
int compSupplyIndex);
61
62     /// @notice Emitted when COMP is distributed t
o a borrower
63     event DistributedBorrowerComp(CToken indexed c
Token, address indexed borrower, uint compDelta, u
int compBorrowIndex);
64
65     /// @notice The threshold above which the flyw
heel transfers COMP, in wei
66     uint public constant compClaimThreshold = 0.00
1e18;
67
68     /// @notice The initial COMP index for a marke
t
69     uint224 public constant compInitialIndex = 1e3
6;
70
71     // closeFactorMantissa must be strictly greate
r than this value
72     uint internal constant closeFactorMinMantissa
= 0.05e18; // 0.05
73
74     // closeFactorMantissa must not exceed this va
lue
75     uint internal constant closeFactorMaxMantissa
= 0.9e18; // 0.9
76
77     // No collateralFactorMantissa may exceed this
value
78     uint internal constant collateralFactorMaxMant
issa = 1.0e18; // 1.0
79
80     // liquidationIncentiveMantissa must be no les
s than this value
81     uint internal constant liquidationIncentiveMin
Mantissa = 1.0e18; // 1.0

```

```

82
83     // liquidationIncentiveMantissa must be no gre
ater than this value
84     uint internal constant liquidationIncentiveMax
Mantissa = 1.5e18; // 1.5
85
86     constructor() public {
87         admin = msg.sender;
88     }
89
90     /** Assets You Are In */
91
92     /**
93      * @notice Returns the assets an account has e
ntered
94      * @param account The address of the account t
o pull assets for
95      * @return A dynamic list with the assets the
account has entered
96      */
97     function getAssetsIn(address account) external
view returns (CToken[] memory) {
98         CToken[] memory assetsIn = accountAssets[a
ccount];
99
100         return assetsIn;
101     }
102
103     /**
104      * @notice Returns whether the given account i
s entered in the given asset
105      * @param account The address of the account t
o check
106      * @param cToken The cToken to check
107      * @return True if the account is in the asse
t, otherwise false.
108      */
109     function checkMembership(address account, CTok
en cToken) external view returns (bool) {
110         return markets[address(cToken)].accountMem
bership[account];
111     }
112
113     /**
114      * @notice Add assets to be included in accoun
t liquidity calculation
115      * @param cTokens The list of addresses of the
cToken markets to be enabled
116      * @return Success indicator for whether each
corresponding market was entered
117      */
118     function enterMarkets(address[] memory cToken
s) public returns (uint[] memory) {
119         uint len = cTokens.length;
120
121         uint[] memory results = new uint[](len);
122         for (uint i = 0; i < len; i++) {
123             CToken cToken = CToken(cTokens[i]);
124
125             results[i] = uint(addToMarketInternal
(cToken, msg.sender));
126         }
127
128         return results;
129     }
130
131     /**

```

```

82
83     // liquidationIncentiveMantissa must be no gre
ater than this value
84     uint internal constant liquidationIncentiveMax
Mantissa = 1.5e18; // 1.5
85
86     constructor() public {
87         admin = msg.sender;
88     }
89
90     /** Assets You Are In */
91
92     /**
93      * @notice Returns the assets an account has e
ntered
94      * @param account The address of the account t
o pull assets for
95      * @return A dynamic list with the assets the
account has entered
96      */
97     function getAssetsIn(address account) external
view returns (CToken[] memory) {
98         CToken[] memory assetsIn = accountAssets[a
ccount];
99
100         return assetsIn;
101     }
102
103     /**
104      * @notice Returns whether the given account i
s entered in the given asset
105      * @param account The address of the account t
o check
106      * @param cToken The cToken to check
107      * @return True if the account is in the asse
t, otherwise false.
108      */
109     function checkMembership(address account, CTok
en cToken) external view returns (bool) {
110         return markets[address(cToken)].accountMem
bership[account];
111     }
112
113     /**
114      * @notice Add assets to be included in accoun
t liquidity calculation
115      * @param cTokens The list of addresses of the
cToken markets to be enabled
116      * @return Success indicator for whether each
corresponding market was entered
117      */
118     function enterMarkets(address[] memory cToken
s) public returns (uint[] memory) {
119         uint len = cTokens.length;
120
121         uint[] memory results = new uint[](len);
122         for (uint i = 0; i < len; i++) {
123             CToken cToken = CToken(cTokens[i]);
124
125             results[i] = uint(addToMarketInternal
(cToken, msg.sender));
126         }
127
128         return results;
129     }
130
131     /**

```

```

132     * @notice Add the market to the borrower's "a
      ssets in" for liquidity calculations
133     * @param cToken The market to enter
134     * @param borrower The address of the account
      to modify
135     * @return Success indicator for whether the m
      arket was entered
136     */
137     function addToMarketInternal(CToken cToken, ad
      dress borrower) internal returns (Error) {
138         Market storage marketToJoin = markets[addr
      ess(cToken)];
139
140         if (!marketToJoin.isListed) {
141             // market is not listed, cannot join
142             return Error.MARKET_NOT_LISTED;
143         }
144
145         if (marketToJoin.accountMembership[borrowe
      r] == true) {
146             // already joined
147             return Error.NO_ERROR;
148         }
149
150         if (accountAssets[borrower].length >= maxA
      ssets) {
151             // no space, cannot join
152             return Error.TOO_MANY_ASSETS;
153         }
154
155         // survived the gauntlet, add to list
156         // NOTE: we store these somewhat redundant
      ly as a significant optimization
157         // this avoids having to iterate through
      the list for the most common use cases
158         // that is, only when we need to perform
      liquidity checks
159         // and not whenever we want to check if a
      n account is in a particular market
160         marketToJoin.accountMembership[borrower] =
      true;
161         accountAssets[borrower].push(cToken);
162
163         emit MarketEntered(cToken, borrower);
164
165         return Error.NO_ERROR;
166     }
167
168     /**
169     * @notice Removes asset from sender's account
      liquidity calculation
170     * @dev Sender must not have an outstanding bo
      rrow balance in the asset,
171     * or be providing necessary collateral for a
      n outstanding borrow.
172     * @param cTokenAddress The address of the ass
      et to be removed
173     * @return Whether or not the account successf
      ully exited the market
174     */
175     function exitMarket(address cTokenAddress) ext
      ernal returns (uint) {
176         CToken cToken = CToken(cTokenAddress);
177         /* Get sender tokensHeld and amountOwed un
      derlying from the cToken */
178         (uint oErr, uint tokensHeld, uint amountOw
      ed, ) = cToken.getAccountSnapshot(msg.sender);

```

```

132     * @notice Add the market to the borrower's "a
      ssets in" for liquidity calculations
133     * @param cToken The market to enter
134     * @param borrower The address of the account
      to modify
135     * @return Success indicator for whether the m
      arket was entered
136     */
137     function addToMarketInternal(CToken cToken, ad
      dress borrower) internal returns (Error) {
138         Market storage marketToJoin = markets[addr
      ess(cToken)];
139
140         if (!marketToJoin.isListed) {
141             // market is not listed, cannot join
142             return Error.MARKET_NOT_LISTED;
143         }
144
145         if (marketToJoin.accountMembership[borrowe
      r] == true) {
146             // already joined
147             return Error.NO_ERROR;
148         }
149
150         if (accountAssets[borrower].length >= maxA
      ssets) {
151             // no space, cannot join
152             return Error.TOO_MANY_ASSETS;
153         }
154
155         // survived the gauntlet, add to list
156         // NOTE: we store these somewhat redundant
      ly as a significant optimization
157         // this avoids having to iterate through
      the list for the most common use cases
158         // that is, only when we need to perform
      liquidity checks
159         // and not whenever we want to check if a
      n account is in a particular market
160         marketToJoin.accountMembership[borrower] =
      true;
161         accountAssets[borrower].push(cToken);
162
163         emit MarketEntered(cToken, borrower);
164
165         return Error.NO_ERROR;
166     }
167
168     /**
169     * @notice Removes asset from sender's account
      liquidity calculation
170     * @dev Sender must not have an outstanding bo
      rrow balance in the asset,
171     * or be providing necessary collateral for a
      n outstanding borrow.
172     * @param cTokenAddress The address of the ass
      et to be removed
173     * @return Whether or not the account successf
      ully exited the market
174     */
175     function exitMarket(address cTokenAddress) ext
      ernal returns (uint) {
176         CToken cToken = CToken(cTokenAddress);
177         /* Get sender tokensHeld and amountOwed un
      derlying from the cToken */
178         (uint oErr, uint tokensHeld, uint amountOw
      ed, ) = cToken.getAccountSnapshot(msg.sender);

```

```

179         require(oErr == 0, "exitMarket: getAccount
Snapshot failed"); // semi-opaque error code
180
181         /* Fail if the sender has a borrow balance
*/
182         if (amountOwed != 0) {
183             return fail(Error.NONZERO_BORROW_BALAN
CE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
184         }
185
186         /* Fail if the sender is not permitted to
redeem all of their tokens */
187         uint allowed = redeemAllowedInternal(cToke
nAddress, msg.sender, tokensHeld);
188         if (allowed != 0) {
189             return failOpaque(Error.REJECTION, Fai
lureInfo.EXIT_MARKET_REJECTION, allowed);
190         }
191
192         Market storage marketToExit = markets[addr
ess(cToken)];
193
194         /* Return true if the sender is not alread
y 'in' the market */
195         if (!marketToExit.accountMembership[msg.se
nder]) {
196             return uint(Error.NO_ERROR);
197         }
198
199         /* Set cToken account membership to false
*/
200         delete marketToExit.accountMembership[msg.
sender];
201
202         /* Delete cToken from the account's list o
f assets */
203         // load into memory for faster iteration
204         CToken[] memory userAssetList = accountAss
ets[msg.sender];
205         uint len = userAssetList.length;
206         uint assetIndex = len;
207         for (uint i = 0; i < len; i++) {
208             if (userAssetList[i] == cToken) {
209                 assetIndex = i;
210                 break;
211             }
212         }
213
214         // We *must* have found the asset in the l
ist or our redundant data structure is broken
215         assert(assetIndex < len);
216
217         // copy last item in list to location of i
tem to be removed, reduce length by 1
218         CToken[] storage storedList = accountAsset
s[msg.sender];
219         storedList[assetIndex] = storedList[stored
List.length - 1];
220         storedList.length--;
221
222         emit MarketExited(cToken, msg.sender);
223
224         return uint(Error.NO_ERROR);
225     }
226
227     /** Policy Hooks */
228
229     /**

```

```

179         require(oErr == 0, "exitMarket: getAccount
Snapshot failed"); // semi-opaque error code
180
181         /* Fail if the sender has a borrow balance
*/
182         if (amountOwed != 0) {
183             return fail(Error.NONZERO_BORROW_BALAN
CE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
184         }
185
186         /* Fail if the sender is not permitted to
redeem all of their tokens */
187         uint allowed = redeemAllowedInternal(cToke
nAddress, msg.sender, tokensHeld);
188         if (allowed != 0) {
189             return failOpaque(Error.REJECTION, Fai
lureInfo.EXIT_MARKET_REJECTION, allowed);
190         }
191
192         Market storage marketToExit = markets[addr
ess(cToken)];
193
194         /* Return true if the sender is not alread
y 'in' the market */
195         if (!marketToExit.accountMembership[msg.se
nder]) {
196             return uint(Error.NO_ERROR);
197         }
198
199         /* Set cToken account membership to false
*/
200         delete marketToExit.accountMembership[msg.
sender];
201
202         /* Delete cToken from the account's list o
f assets */
203         // load into memory for faster iteration
204         CToken[] memory userAssetList = accountAss
ets[msg.sender];
205         uint len = userAssetList.length;
206         uint assetIndex = len;
207         for (uint i = 0; i < len; i++) {
208             if (userAssetList[i] == cToken) {
209                 assetIndex = i;
210                 break;
211             }
212         }
213
214         // We *must* have found the asset in the l
ist or our redundant data structure is broken
215         assert(assetIndex < len);
216
217         // copy last item in list to location of i
tem to be removed, reduce length by 1
218         CToken[] storage storedList = accountAsset
s[msg.sender];
219         storedList[assetIndex] = storedList[stored
List.length - 1];
220         storedList.length--;
221
222         emit MarketExited(cToken, msg.sender);
223
224         return uint(Error.NO_ERROR);
225     }
226
227     /** Policy Hooks */
228
229     /**

```

```

230     * @notice Checks if the account should be all
owed to mint tokens in the given market
231     * @param cToken The market to verify the mint
against
232     * @param minter The account which would get t
he minted tokens
233     * @param mintAmount The amount of underlying
being supplied to the market in exchange for toke
ns
234     * @return 0 if the mint is allowed, otherwise
a semi-opaque error code (See ErrorReporter.sol)
235     */
236     function mintAllowed(address cToken, address m
inter, uint mintAmount) external returns (uint) {
237         // Pausing is a very serious situation - w
e revert to sound the alarms
238         require(!mintGuardianPaused[cToken], "mint
is paused");
239
240         // Shh - currently unused
241         minter;
242         mintAmount;
243
244         if (!markets[cToken].isListed) {
245             return uint(Error.MARKET_NOT_LISTED);
246         }
247
248         // Keep the flywheel moving
249         updateCompSupplyIndex(cToken);
250         distributeSupplierComp(cToken, minter, fal
se);
251
252         return uint(Error.NO_ERROR);
253     }
254
255     /**
256     * @notice Validates mint and reverts on rejec
tion. May emit logs.
257     * @param cToken Asset being minted
258     * @param minter The address minting the token
s
259     * @param actualMintAmount The amount of the u
nderlying asset being minted
260     * @param mintTokens The number of tokens bein
g minted
261     */
262     function mintVerify(address cToken, address mi
nter, uint actualMintAmount, uint mintTokens) exte
rnal {
263         // Shh - currently unused
264         cToken;
265         minter;
266         actualMintAmount;
267         mintTokens;
268
269         // Shh - we don't ever want this hook to b
e marked pure
270         if (false) {
271             maxAssets = maxAssets;
272         }
273     }
274
275     /**

```

```

230     * @notice Checks if the account should be all
owed to mint tokens in the given market
231     * @param cToken The market to verify the mint
against
232     * @param minter The account which would get t
he minted tokens
233     * @param mintAmount The amount of underlying
being supplied to the market in exchange for toke
ns
234     * @return 0 if the mint is allowed, otherwise
a semi-opaque error code (See ErrorReporter.sol)
235     */
236     function mintAllowed(address cToken, address m
inter, uint mintAmount) external returns (uint) {
237         // Pausing is a very serious situation - w
e revert to sound the alarms
238         require(!mintGuardianPaused[cToken], "mint
is paused");
239
240         // Shh - currently unused
241         minter;
242         mintAmount;
243
244         if (!markets[cToken].isListed) {
245             return uint(Error.MARKET_NOT_LISTED);
246         }
247
248         // update the asset price
249         oracle.updatePrice(CToken(cToken));
250
251         // Keep the flywheel moving
252         updateCompSupplyIndex(cToken);
253         distributeSupplierComp(cToken, minter, fal
se);
254
255         return uint(Error.NO_ERROR);
256     }
257
258     /**
259     * @notice Validates mint and reverts on rejec
tion. May emit logs.
260     * @param cToken Asset being minted
261     * @param minter The address minting the token
s
262     * @param actualMintAmount The amount of the u
nderlying asset being minted
263     * @param mintTokens The number of tokens bein
g minted
264     */
265     function mintVerify(address cToken, address mi
nter, uint actualMintAmount, uint mintTokens) exte
rnal {
266         // Shh - currently unused
267         cToken;
268         minter;
269         actualMintAmount;
270         mintTokens;
271
272         // Shh - we don't ever want this hook to b
e marked pure
273         if (false) {
274             maxAssets = maxAssets;
275         }
276     }
277
278     /**

```

```

276     * @notice Checks if the account should be all
owed to redeem tokens in the given market
277     * @param cToken The market to verify the rede
em against
278     * @param redeemer The account which would red
eem the tokens
279     * @param redeemTokens The number of cTokens t
o exchange for the underlying asset in the market
280     * @return 0 if the redeem is allowed, otherwi
se a semi-opaque error code (See ErrorReporter.so
l)
281     */
282     function redeemAllowed(address cToken, address
redeemer, uint redeemTokens) external returns (uin
t) {
283         uint allowed = redeemAllowedInternal(cToke
n, redeemer, redeemTokens);
284         if (allowed != uint(Error.NO_ERROR)) {
285             return allowed;
286         }
287         // Keep the flywheel moving
288         updateCompSupplyIndex(cToken);
289         distributeSupplierComp(cToken, redeemer, f
alse);
290     }
291     return uint(Error.NO_ERROR);
292 }
293
294 function redeemAllowedInternal(address cToken,
address redeemer, uint redeemTokens) internal view
returns (uint) {
295     if (!markets[cToken].isListed) {
296         return uint(Error.MARKET_NOT_LISTED);
297     }
298     /* If the redeemer is not 'in' the market,
then we can bypass the liquidity check */
299     if (!markets[cToken].accountMembership[red
eemer]) {
300         return uint(Error.NO_ERROR);
301     }
302     /* Otherwise, perform a hypothetical liqui
dity check to guard against shortfall */
303     (Error err, , uint shortfall) = getHypothe
ticalAccountLiquidityInternal(redeemer, CToken(cTo
ken), redeemTokens, 0);
304     if (err != Error.NO_ERROR) {
305         return uint(err);
306     }
307     if (shortfall > 0) {
308         return uint(Error.INSUFFICIENT_LIQUIDI
TY);
309     }
310     return uint(Error.NO_ERROR);
311 }
312
313 /**
314     * @notice Validates redeem and reverts on rej
ection. May emit logs.
315     * @param cToken Asset being redeemed

```

```

279     * @notice Checks if the account should be all
owed to redeem tokens in the given market
280     * @param cToken The market to verify the rede
em against
281     * @param redeemer The account which would red
eem the tokens
282     * @param redeemTokens The number of cTokens t
o exchange for the underlying asset in the market
283     * @return 0 if the redeem is allowed, otherwi
se a semi-opaque error code (See ErrorReporter.so
l)
284     */
285     function redeemAllowed(address cToken, address
redeemer, uint redeemTokens) external returns (uin
t) {
286         // update the asset price
287         oracle.updatePrice(CToken(cToken));
288     }
289     uint allowed = redeemAllowedInternal(cToke
n, redeemer, redeemTokens);
290     if (allowed != uint(Error.NO_ERROR)) {
291         return allowed;
292     }
293     // Keep the flywheel moving
294     updateCompSupplyIndex(cToken);
295     distributeSupplierComp(cToken, redeemer, f
alse);
296     return uint(Error.NO_ERROR);
297 }
298
299 function redeemAllowedInternal(address cToken,
address redeemer, uint redeemTokens) internal view
returns (uint) {
300     if (!markets[cToken].isListed) {
301         return uint(Error.MARKET_NOT_LISTED);
302     }
303     /* If the redeemer is not 'in' the market,
then we can bypass the liquidity check */
304     if (!markets[cToken].accountMembership[red
eemer]) {
305         return uint(Error.NO_ERROR);
306     }
307     /* Otherwise, perform a hypothetical liqui
dity check to guard against shortfall */
308     (Error err, , uint shortfall) = getHypothe
ticalAccountLiquidityInternal(redeemer, CToken(cTo
ken), redeemTokens, 0);
309     if (err != Error.NO_ERROR) {
310         return uint(err);
311     }
312     if (shortfall > 0) {
313         return uint(Error.INSUFFICIENT_LIQUIDI
TY);
314     }
315     return uint(Error.NO_ERROR);
316 }
317
318 /**
319     * @notice Validates redeem and reverts on rej
ection. May emit logs.
320     * @param cToken Asset being redeemed

```



```

320     * @param redeemer The address redeeming the t
    oks
321     * @param redeemAmount The amount of the under
    lying asset being redeemed
322     * @param redeemTokens The number of tokens be
    ing redeemed
323     */
324     function redeemVerify(address cToken, address
    redeemer, uint redeemAmount, uint redeemTokens) e
    xternal {
325         // Shh - currently unused
326         cToken;
327         redeemer;
328
329         // Require tokens is zero or amount is als
    o zero
330         if (redeemTokens == 0 && redeemAmount > 0)
    {
331             revert("redeemTokens zero");
332         }
333     }
334
335     /**
336     * @notice Checks if the account should be all
    owed to borrow the underlying asset of the given m
    arket
337     * @param cToken The market to verify the borr
    ow against
338     * @param borrower The account which would bor
    row the asset
339     * @param borrowAmount The amount of underlyin
    g the account would borrow
340     * @return 0 if the borrow is allowed, otherwi
    se a semi-opaque error code (See ErrorReporter.so
    l)
341     */
342     function borrowAllowed(address cToken, address
    borrower, uint borrowAmount) external returns (uin
    t) {
343         // Pausing is a very serious situation - w
    e revert to sound the alarms
344         require(!borrowGuardianPaused[cToken], "bo
    rrow is paused");
345
346         if (!markets[cToken].isListed) {
347             return uint(Error.MARKET_NOT_LISTED);
348         }
349
350         if (!markets[cToken].accountMembership[bor
    rower]) {
351             // only cTokens may call borrowAllowed
    if borrower not in market
352             require(msg.sender == cToken, "sender
    must be cToken");
353
354             // attempt to add borrower to the mark
    et
355             Error err = addToMarketInternal(cToken
    (msg.sender), borrower);
356             if (err != Error.NO_ERROR) {
357                 return uint(err);
358             }
359
360             // it should be impossible to break th
    e important invariant

```

```

326     * @param redeemer The address redeeming the t
    oks
327     * @param redeemAmount The amount of the under
    lying asset being redeemed
328     * @param redeemTokens The number of tokens be
    ing redeemed
329     */
330     function redeemVerify(address cToken, address
    redeemer, uint redeemAmount, uint redeemTokens) e
    xternal {
331         // Shh - currently unused
332         cToken;
333         redeemer;
334
335         // Require tokens is zero or amount is als
    o zero
336         if (redeemTokens == 0 && redeemAmount > 0)
    {
337             revert("redeemTokens zero");
338         }
339     }
340
341     /**
342     * @notice Checks if the account should be all
    owed to borrow the underlying asset of the given m
    arket
343     * @param cToken The market to verify the borr
    ow against
344     * @param borrower The account which would bor
    row the asset
345     * @param borrowAmount The amount of underlyin
    g the account would borrow
346     * @return 0 if the borrow is allowed, otherwi
    se a semi-opaque error code (See ErrorReporter.so
    l)
347     */
348     function borrowAllowed(address cToken, address
    borrower, uint borrowAmount) external returns (uin
    t) {
349         // Pausing is a very serious situation - w
    e revert to sound the alarms
350         require(!borrowGuardianPaused[cToken], "bo
    rrow is paused");
351
352         if (!markets[cToken].isListed) {
353             return uint(Error.MARKET_NOT_LISTED);
354         }
355
356         if (!markets[cToken].accountMembership[bor
    rower]) {
357             // only cTokens may call borrowAllowed
    if borrower not in market
358             require(msg.sender == cToken, "sender
    must be cToken");
359
360             // attempt to add borrower to the mark
    et
361             Error err = addToMarketInternal(cToken
    (msg.sender), borrower);
362             if (err != Error.NO_ERROR) {
363                 return uint(err);
364             }
365
366             // it should be impossible to break th
    e important invariant

```



```

361         assert(markets[cToken].accountMembersh
ip[borrower]);
362     }
363
364     if (oracle.getUnderlyingPrice(CToken(cToke
n)) == 0) {
365         return uint(Error.PRICE_ERROR);
366     }
367
368     (Error err, , uint shortfall) = getHypothe
ticalAccountLiquidityInternal(borrower, CToken(cTo
ken), 0, borrowAmount);
369     if (err != Error.NO_ERROR) {
370         return uint(err);
371     }
372     if (shortfall > 0) {
373         return uint(Error.INSUFFICIENT_LIQUIDI
TY);
374     }
375
376     // Keep the flywheel moving
377     Exp memory borrowIndex = Exp({mantissa: CT
oken(cToken).borrowIndex()});
378     updateCompBorrowIndex(cToken, borrowInde
x);
379     distributeBorrowerComp(cToken, borrower, b
orrowIndex, false);
380
381     return uint(Error.NO_ERROR);
382 }
383
384 /**
385  * @notice Validates borrow and reverts on rej
ection. May emit logs.
386  * @param cToken Asset whose underlying is bei
ng borrowed
387  * @param borrower The address borrowing the u
nderlying
388  * @param borrowAmount The amount of the under
lying asset requested to borrow
389  */
390 function borrowVerify(address cToken, address
borrower, uint borrowAmount) external {
391     // Shh - currently unused
392     cToken;
393     borrower;
394     borrowAmount;
395
396     // Shh - we don't ever want this hook to b
e marked pure
397     if (false) {
398         maxAssets = maxAssets;
399     }
400 }
401
402 /**
403  * @notice Checks if the account should be all
owed to repay a borrow in the given market
404  * @param cToken The market to verify the repa
y against
405  * @param payer The account which would repay
the asset
406  * @param borrower The account which would bor
rowed the asset

```

```

367         assert(markets[cToken].accountMembersh
ip[borrower]);
368     }
369
370     // update the asset price
371     oracle.updatePrice(CToken(cToken));
372
373     if (oracle.getUnderlyingPrice(CToken(cToke
n)) == 0) {
374         return uint(Error.PRICE_ERROR);
375     }
376
377     (Error err, , uint shortfall) = getHypothe
ticalAccountLiquidityInternal(borrower, CToken(cTo
ken), 0, borrowAmount);
378     if (err != Error.NO_ERROR) {
379         return uint(err);
380     }
381     if (shortfall > 0) {
382         return uint(Error.INSUFFICIENT_LIQUIDI
TY);
383     }
384
385     // Keep the flywheel moving
386     Exp memory borrowIndex = Exp({mantissa: CT
oken(cToken).borrowIndex()});
387     updateCompBorrowIndex(cToken, borrowInde
x);
388     distributeBorrowerComp(cToken, borrower, b
orrowIndex, false);
389
390     return uint(Error.NO_ERROR);
391 }
392
393 /**
394  * @notice Validates borrow and reverts on rej
ection. May emit logs.
395  * @param cToken Asset whose underlying is bei
ng borrowed
396  * @param borrower The address borrowing the u
nderlying
397  * @param borrowAmount The amount of the under
lying asset requested to borrow
398  */
399 function borrowVerify(address cToken, address
borrower, uint borrowAmount) external {
400     // Shh - currently unused
401     cToken;
402     borrower;
403     borrowAmount;
404
405     // Shh - we don't ever want this hook to b
e marked pure
406     if (false) {
407         maxAssets = maxAssets;
408     }
409 }
410
411 /**
412  * @notice Checks if the account should be all
owed to repay a borrow in the given market
413  * @param cToken The market to verify the repa
y against
414  * @param payer The account which would repay
the asset
415  * @param borrower The account which would bor
rowed the asset

```

```

407     * @param repayAmount The amount of the underl
ying asset the account would repay
408     * @return 0 if the repay is allowed, otherwis
e a semi-opaque error code (See ErrorReporter.sol)
409     */
410     function repayBorrowAllowed(
411         address cToken,
412         address payer,
413         address borrower,
414         uint repayAmount) external returns (uint)
415     {
416         // Shh - currently unused
417         payer;
418         borrower;
419         repayAmount;
420         if (!markets[cToken].isListed) {
421             return uint(Error.MARKET_NOT_LISTED);
422         }
423
424         // Keep the flywheel moving
425         Exp memory borrowIndex = Exp({mantissa: CT
oken(cToken).borrowIndex()});
426         updateCompBorrowIndex(cToken, borrowInde
x);
427         distributeBorrowerComp(cToken, borrower, b
orrowIndex, false);
428
429         return uint(Error.NO_ERROR);
430     }
431
432     /**
433     * @notice Validates repayBorrow and reverts o
n rejection. May emit logs.
434     * @param cToken Asset being repaid
435     * @param payer The address repaying the borro
w
436     * @param borrower The address of the borrower
437     * @param actualRepayAmount The amount of unde
rlying being repaid
438     */
439     function repayBorrowVerify(
440         address cToken,
441         address payer,
442         address borrower,
443         uint actualRepayAmount,
444         uint borrowerIndex) external {
445         // Shh - currently unused
446         cToken;
447         payer;
448         borrower;
449         actualRepayAmount;
450         borrowerIndex;
451
452         // Shh - we don't ever want this hook to b
e marked pure
453         if (false) {
454             maxAssets = maxAssets;
455         }
456     }
457
458     /**
459     * @notice Checks if the liquidation should be
allowed to occur

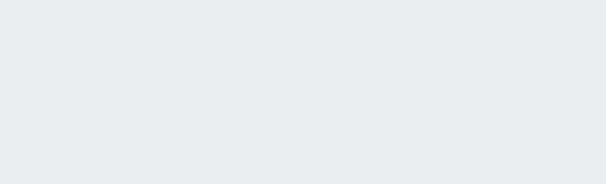
```

```

416     * @param repayAmount The amount of the underl
ying asset the account would repay
417     * @return 0 if the repay is allowed, otherwis
e a semi-opaque error code (See ErrorReporter.sol)
418     */
419     function repayBorrowAllowed(
420         address cToken,
421         address payer,
422         address borrower,
423         uint repayAmount) external returns (uint)
424     {
425         // Shh - currently unused
426         payer;
427         borrower;
428         repayAmount;
429         if (!markets[cToken].isListed) {
430             return uint(Error.MARKET_NOT_LISTED);
431         }
432
433         // update the asset price
434         oracle.updatePrice(CToken(cToken));
435
436         // Keep the flywheel moving
437         Exp memory borrowIndex = Exp({mantissa: CT
oken(cToken).borrowIndex()});
438         updateCompBorrowIndex(cToken, borrowInde
x);
439         distributeBorrowerComp(cToken, borrower, b
orrowIndex, false);
440
441         return uint(Error.NO_ERROR);
442     }
443
444     /**
445     * @notice Validates repayBorrow and reverts o
n rejection. May emit logs.
446     * @param cToken Asset being repaid
447     * @param payer The address repaying the borro
w
448     * @param borrower The address of the borrower
449     * @param actualRepayAmount The amount of unde
rlying being repaid
450     */
451     function repayBorrowVerify(
452         address cToken,
453         address payer,
454         address borrower,
455         uint actualRepayAmount,
456         uint borrowerIndex) external {
457         // Shh - currently unused
458         cToken;
459         payer;
460         borrower;
461         actualRepayAmount;
462         borrowerIndex;
463
464         // Shh - we don't ever want this hook to b
e marked pure
465         if (false) {
466             maxAssets = maxAssets;
467         }
468     }
469
470     /**
471     * @notice Checks if the liquidation should be
allowed to occur

```

```

460     * @param cTokenBorrowed Asset which was borro
wed by the borrower
461     * @param cTokenCollateral Asset which was use
d as collateral and will be seized
462     * @param liquidator The address repaying the
borrow and seizing the collateral
463     * @param borrower The address of the borrower
464     * @param repayAmount The amount of underlying
being repaid
465     */
466     function liquidateBorrowAllowed(
467         address cTokenBorrowed,
468         address cTokenCollateral,
469         address liquidator,
470         address borrower,
471         uint repayAmount) external returns (uint)
    {
472         // Shh - currently unused
473         liquidator;
474
475         if (!markets[cTokenBorrowed].isListed || !
markets[cTokenCollateral].isListed) {
476             return uint(Error.MARKET_NOT_LISTED);
477         }
478

479         /* The borrower must have shortfall in ord
er to be liquidatable */
480         (Error err, , uint shortfall) = getAccount
LiquidityInternal(borrower);
481         if (err != Error.NO_ERROR) {
482             return uint(err);
483         }
484         if (shortfall == 0) {
485             return uint(Error.INSUFFICIENT_SHORTFA
LL);
486         }
487
488         /* The liquidator may not repay more than
what is allowed by the closeFactor */
489         uint borrowBalance = CToken(cTokenBorrowe
d).borrowBalanceStored(borrower);
490         (MathError mathErr, uint maxClose) = mulSc
alarTruncate(Exp({mantissa: closeFactorMantissa}),
borrowBalance);
491         if (mathErr != MathError.NO_ERROR) {
492             return uint(Error.MATH_ERROR);
493         }
494         if (repayAmount > maxClose) {
495             return uint(Error.TOO_MUCH_REPAY);
496         }
497
498         return uint(Error.NO_ERROR);
499     }
500
501     /**
502     * @notice Validates liquidateBorrow and rever
ts on rejection. May emit logs.
503     * @param cTokenBorrowed Asset which was borro
wed by the borrower

```

```

472     * @param cTokenBorrowed Asset which was borro
wed by the borrower
473     * @param cTokenCollateral Asset which was use
d as collateral and will be seized
474     * @param liquidator The address repaying the
borrow and seizing the collateral
475     * @param borrower The address of the borrower
476     * @param repayAmount The amount of underlying
being repaid
477     */
478     function liquidateBorrowAllowed(
479         address cTokenBorrowed,
480         address cTokenCollateral,
481         address liquidator,
482         address borrower,
483         uint repayAmount) external returns (uint)
    {
484         // Shh - currently unused
485         liquidator;
486
487         if (!markets[cTokenBorrowed].isListed || !
markets[cTokenCollateral].isListed) {
488             return uint(Error.MARKET_NOT_LISTED);
489         }
490
491         // update the asset price
492         oracle.updatePrice(CToken(cTokenBorrowe
d));
493         oracle.updatePrice(CToken(cTokenCollatera
l));
494
495         /* The borrower must have shortfall in ord
er to be liquidatable */
496         (Error err, , uint shortfall) = getAccount
LiquidityInternal(borrower);
497         if (err != Error.NO_ERROR) {
498             return uint(err);
499         }
500         if (shortfall == 0) {
501             return uint(Error.INSUFFICIENT_SHORTFA
LL);
502         }
503
504         /* The liquidator may not repay more than
what is allowed by the closeFactor */
505         uint borrowBalance = CToken(cTokenBorrowe
d).borrowBalanceStored(borrower);
506         (MathError mathErr, uint maxClose) = mulSc
alarTruncate(Exp({mantissa: closeFactorMantissa}),
borrowBalance);
507         if (mathErr != MathError.NO_ERROR) {
508             return uint(Error.MATH_ERROR);
509         }
510         if (repayAmount > maxClose) {
511             return uint(Error.TOO_MUCH_REPAY);
512         }
513
514         return uint(Error.NO_ERROR);
515     }
516
517     /**
518     * @notice Validates liquidateBorrow and rever
ts on rejection. May emit logs.
519     * @param cTokenBorrowed Asset which was borro
wed by the borrower

```

```

504     * @param cTokenCollateral Asset which was use
    d as collateral and will be seized
505     * @param liquidator The address repaying the
    borrow and seizing the collateral
506     * @param borrower The address of the borrower
507     * @param actualRepayAmount The amount of unde
    rlying being repaid
508     */
509     function liquidateBorrowVerify(
510         address cTokenBorrowed,
511         address cTokenCollateral,
512         address liquidator,
513         address borrower,
514         uint actualRepayAmount,
515         uint seizeTokens) external {
516         // Shh - currently unused
517         cTokenBorrowed;
518         cTokenCollateral;
519         liquidator;
520         borrower;
521         actualRepayAmount;
522         seizeTokens;
523
524         // Shh - we don't ever want this hook to b
    e marked pure
525         if (false) {
526             maxAssets = maxAssets;
527         }
528     }
529
530     /**
531     * @notice Checks if the seizing of assets sho
    uld be allowed to occur
532     * @param cTokenCollateral Asset which was use
    d as collateral and will be seized
533     * @param cTokenBorrowed Asset which was borro
    wed by the borrower
534     * @param liquidator The address repaying the
    borrow and seizing the collateral
535     * @param borrower The address of the borrower
536     * @param seizeTokens The number of collateral
    tokens to seize
537     */
538     function seizeAllowed(
539         address cTokenCollateral,
540         address cTokenBorrowed,
541         address liquidator,
542         address borrower,
543         uint seizeTokens) external returns (uint)
    {
544         // Pausing is a very serious situation - w
    e revert to sound the alarms
545         require(!seizeGuardianPaused, "seize is pa
    used");
546
547         // Shh - currently unused
548         seizeTokens;
549
550         if (!markets[cTokenCollateral].isListed ||
    !markets[cTokenBorrowed].isListed) {
551             return uint(Error.MARKET_NOT_LISTED);
552         }
553
554         if (CToken(cTokenCollateral).comptroller()
    != CToken(cTokenBorrowed).comptroller()) {

```

```

520     * @param cTokenCollateral Asset which was use
    d as collateral and will be seized
521     * @param liquidator The address repaying the
    borrow and seizing the collateral
522     * @param borrower The address of the borrower
523     * @param actualRepayAmount The amount of unde
    rlying being repaid
524     */
525     function liquidateBorrowVerify(
526         address cTokenBorrowed,
527         address cTokenCollateral,
528         address liquidator,
529         address borrower,
530         uint actualRepayAmount,
531         uint seizeTokens) external {
532         // Shh - currently unused
533         cTokenBorrowed;
534         cTokenCollateral;
535         liquidator;
536         borrower;
537         actualRepayAmount;
538         seizeTokens;
539
540         // Shh - we don't ever want this hook to b
    e marked pure
541         if (false) {
542             maxAssets = maxAssets;
543         }
544     }
545
546     /**
547     * @notice Checks if the seizing of assets sho
    uld be allowed to occur
548     * @param cTokenCollateral Asset which was use
    d as collateral and will be seized
549     * @param cTokenBorrowed Asset which was borro
    wed by the borrower
550     * @param liquidator The address repaying the
    borrow and seizing the collateral
551     * @param borrower The address of the borrower
552     * @param seizeTokens The number of collateral
    tokens to seize
553     */
554     function seizeAllowed(
555         address cTokenCollateral,
556         address cTokenBorrowed,
557         address liquidator,
558         address borrower,
559         uint seizeTokens) external returns (uint)
    {
560         // Pausing is a very serious situation - w
    e revert to sound the alarms
561         require(!seizeGuardianPaused, "seize is pa
    used");
562
563         // Shh - currently unused
564         seizeTokens;
565
566         if (!markets[cTokenCollateral].isListed ||
    !markets[cTokenBorrowed].isListed) {
567             return uint(Error.MARKET_NOT_LISTED);
568         }
569
570         if (CToken(cTokenCollateral).comptroller()
    != CToken(cTokenBorrowed).comptroller()) {

```

```

555         return uint(Error.COMPTROLLER_MISMATCH
H);
556     }

```

```

557
558     // Keep the flywheel moving
559     updateCompSupplyIndex(cTokenCollateral);
560     distributeSupplierComp(cTokenCollateral, borrower, false);
561     distributeSupplierComp(cTokenCollateral, liquidator, false);
562
563     return uint(Error.NO_ERROR);
564 }
565
566 /**
567  * @notice Validates seize and reverts on rejection. May emit logs.
568  * @param cTokenCollateral Asset which was used as collateral and will be seized
569  * @param cTokenBorrowed Asset which was borrowed by the borrower
570  * @param liquidator The address repaying the borrow and seizing the collateral
571  * @param borrower The address of the borrower
572  * @param seizeTokens The number of collateral tokens to seize
573  */
574 function seizeVerify(
575     address cTokenCollateral,
576     address cTokenBorrowed,
577     address liquidator,
578     address borrower,
579     uint seizeTokens) external {
580     // Shh - currently unused
581     cTokenCollateral;
582     cTokenBorrowed;
583     liquidator;
584     borrower;
585     seizeTokens;
586
587     // Shh - we don't ever want this hook to be marked pure
588     if (false) {
589         maxAssets = maxAssets;
590     }
591 }
592
593 /**
594  * @notice Checks if the account should be allowed to transfer tokens in the given market
595  * @param cToken The market to verify the transfer against
596  * @param src The account which sources the tokens
597  * @param dst The account which receives the tokens
598  * @param transferTokens The number of cTokens to transfer

```

```

571         return uint(Error.COMPTROLLER_MISMATCH
H);
572     }

```

```

573
574     // update the asset price
575     oracle.updatePrice(cToken(cTokenCollateral));
576     oracle.updatePrice(cToken(cTokenBorrowed));
577
578     // Keep the flywheel moving
579     updateCompSupplyIndex(cTokenCollateral);
580     distributeSupplierComp(cTokenCollateral, borrower, false);
581     distributeSupplierComp(cTokenCollateral, liquidator, false);
582
583     return uint(Error.NO_ERROR);
584 }
585
586 /**
587  * @notice Validates seize and reverts on rejection. May emit logs.
588  * @param cTokenCollateral Asset which was used as collateral and will be seized
589  * @param cTokenBorrowed Asset which was borrowed by the borrower
590  * @param liquidator The address repaying the borrow and seizing the collateral
591  * @param borrower The address of the borrower
592  * @param seizeTokens The number of collateral tokens to seize
593  */
594 function seizeVerify(
595     address cTokenCollateral,
596     address cTokenBorrowed,
597     address liquidator,
598     address borrower,
599     uint seizeTokens) external {
600     // Shh - currently unused
601     cTokenCollateral;
602     cTokenBorrowed;
603     liquidator;
604     borrower;
605     seizeTokens;
606
607     // Shh - we don't ever want this hook to be marked pure
608     if (false) {
609         maxAssets = maxAssets;
610     }
611 }
612
613 /**
614  * @notice Checks if the account should be allowed to transfer tokens in the given market
615  * @param cToken The market to verify the transfer against
616  * @param src The account which sources the tokens
617  * @param dst The account which receives the tokens
618  * @param transferTokens The number of cTokens to transfer

```

```

599     * @return 0 if the transfer is allowed, other
      wise a semi-opaque error code (See ErrorReporter.s
      ol)
600     */
601     function transferAllowed(address cToken, addre
      ss src, address dst, uint transferTokens) external
      returns (uint) {
602         // Pausing is a very serious situation - w
      e revert to sound the alarms
603         require(!transferGuardianPaused, "transfer
      is paused");
604
605         // Currently the only consideration is whe
      ther or not
606         // the src is allowed to redeem this many
      tokens
607         uint allowed = redeemAllowedInternal(cToke
      n, src, transferTokens);
608         if (allowed != uint(Error.NO_ERROR)) {
609             return allowed;
610         }
611         // Keep the flywheel moving
612         updateCompSupplyIndex(cToken);
613         distributeSupplierComp(cToken, src, fals
      e);
614         distributeSupplierComp(cToken, dst, fals
      e);
615         return uint(Error.NO_ERROR);
616     }
617     /**
618     * @notice Validates transfer and reverts on r
      ejection. May emit logs.
619     * @param cToken Asset being transferred
620     * @param src The account which sources the to
      kens
621     * @param dst The account which receives the t
      okens
622     * @param transferTokens The number of cTokens
      to transfer
623     */
624     function transferVerify(address cToken, addres
      s src, address dst, uint transferTokens) external
      {
625         // Shh - currently unused
626         cToken;
627         src;
628         dst;
629         transferTokens;
630         // Shh - we don't ever want this hook to b
      e marked pure
631         if (false) {
632             maxAssets = maxAssets;
633         }
634     }
635     /** Liquidity/Liquidation Calculations */
636     /**
637     * @dev Local vars for avoiding stack-depth li
      mits in calculating account liquidity.

```

```

619     * @return 0 if the transfer is allowed, other
      wise a semi-opaque error code (See ErrorReporter.s
      ol)
620     */
621     function transferAllowed(address cToken, addre
      ss src, address dst, uint transferTokens) external
      returns (uint) {
622         // Pausing is a very serious situation - w
      e revert to sound the alarms
623         require(!transferGuardianPaused, "transfer
      is paused");
624
625         // update the asset price
626         oracle.updatePrice(cToken(cToken));
627
628         // Currently the only consideration is whe
      ther or not
629         // the src is allowed to redeem this many
      tokens
630         uint allowed = redeemAllowedInternal(cToke
      n, src, transferTokens);
631         if (allowed != uint(Error.NO_ERROR)) {
632             return allowed;
633         }
634         // Keep the flywheel moving
635         updateCompSupplyIndex(cToken);
636         distributeSupplierComp(cToken, src, fals
      e);
637         distributeSupplierComp(cToken, dst, fals
      e);
638         return uint(Error.NO_ERROR);
639     }
640     /**
641     * @notice Validates transfer and reverts on r
      ejection. May emit logs.
642     * @param cToken Asset being transferred
643     * @param src The account which sources the to
      kens
644     * @param dst The account which receives the t
      okens
645     * @param transferTokens The number of cTokens
      to transfer
646     */
647     function transferVerify(address cToken, addres
      s src, address dst, uint transferTokens) external
      {
648         // Shh - currently unused
649         cToken;
650         src;
651         dst;
652         transferTokens;
653         // Shh - we don't ever want this hook to b
      e marked pure
654         if (false) {
655             maxAssets = maxAssets;
656         }
657     }
658     /** Liquidity/Liquidation Calculations */
659     /**
660     * @dev Local vars for avoiding stack-depth li
      mits in calculating account liquidity.

```

```

644     * Note that `cTokenBalance` is the number of
        cTokens the account owns in the market,
645     * whereas `borrowBalance` is the amount of u
        nderlying that the account has borrowed.
646     */
647     struct AccountLiquidityLocalVars {
648         uint sumCollateral;
649         uint sumBorrowPlusEffects;
650         uint cTokenBalance;
651         uint borrowBalance;
652         uint exchangeRateMantissa;
653         uint oraclePriceMantissa;
654         Exp collateralFactor;
655         Exp exchangeRate;
656         Exp oraclePrice;
657         Exp tokensToDenom;
658     }
659     /**
660     * @notice Determine the current account liqui
        dity wrt collateral requirements
661     * @return (possible error code (semi-opaque),
        account liquidity in excess of col
662     lateral requirements,
663     *         account shortfall below collateral
        requirements)
664     */
665     function getAccountLiquidity(address account)
        public view returns (uint, uint, uint) {
666         (Error err, uint liquidity, uint shortfal
        l) = getHypotheticalAccountLiquidityInternal(accou
        nt, CToken(0), 0, 0);
667         return (uint(err), liquidity, shortfall);
668     }
669     /**
670     * @notice Determine the current account liqui
        dity wrt collateral requirements
671     * @return (possible error code,
672     account liquidity in excess of col
673     lateral requirements,
674     *         account shortfall below collateral
        requirements)
675     */
676     function getAccountLiquidityInternal(address a
        ccount) internal view returns (Error, uint, uint)
        {
677         return getHypotheticalAccountLiquidityInte
        rnal(account, CToken(0), 0, 0);
678     }
679     /**
680     * @notice Determine what the account liquidit
        y would be if the given amounts were redeemed/borr
        owed
681     * @param cTokenModify The market to hypotheti
        cally redeem/borrow in
682     * @param account The account to determine liq
        uidity for
683     * @param redeemTokens The number of tokens to
        hypothetically redeem
684     * @param borrowAmount The amount of underlyin
        g to hypothetically borrow
685     * @return (possible error code (semi-opaque),

```

```

667     * Note that `cTokenBalance` is the number of
        cTokens the account owns in the market,
668     * whereas `borrowBalance` is the amount of u
        nderlying that the account has borrowed.
669     */
670     struct AccountLiquidityLocalVars {
671         uint sumCollateral;
672         uint sumBorrowPlusEffects;
673         uint cTokenBalance;
674         uint borrowBalance;
675         uint exchangeRateMantissa;
676         uint oraclePriceMantissa;
677         Exp collateralFactor;
678         Exp exchangeRate;
679         Exp oraclePrice;
680         Exp tokensToDenom;
681     }
682     /**
683     * @notice Determine the current account liqui
        dity wrt collateral requirements
684     * @return (possible error code (semi-opaque),
        account liquidity in excess of col
685     lateral requirements,
686     *         account shortfall below collateral
        requirements)
687     */
688     function getAccountLiquidity(address account)
        public view returns (uint, uint, uint) {
689         (Error err, uint liquidity, uint shortfal
        l) = getHypotheticalAccountLiquidityInternal(accou
        nt, CToken(0), 0, 0);
690         return (uint(err), liquidity, shortfall);
691     }
692     /**
693     * @notice Determine the current account liqui
        dity wrt collateral requirements
694     * @return (possible error code,
695     account liquidity in excess of col
696     lateral requirements,
697     *         account shortfall below collateral
        requirements)
698     */
699     function getAccountLiquidityInternal(address a
        ccount) internal view returns (Error, uint, uint)
        {
700         return getHypotheticalAccountLiquidityInte
        rnal(account, CToken(0), 0, 0);
701     }
702     /**
703     * @notice Determine what the account liquidit
        y would be if the given amounts were redeemed/borr
        owed
704     * @param cTokenModify The market to hypotheti
        cally redeem/borrow in
705     * @param account The account to determine liq
        uidity for
706     * @param redeemTokens The number of tokens to
        hypothetically redeem
707     * @param borrowAmount The amount of underlyin
        g to hypothetically borrow
708     * @return (possible error code (semi-opaque),

```



```

689             hypothetical account liquidity in
              excess of collateral requirements,
690     *           hypothetical account shortfall bel
ow collateral requirements)
691     */
692     function getHypotheticalAccountLiquidity(
693         address account,
694         address cTokenModify,
695         uint redeemTokens,
696         uint borrowAmount) public view returns (ui
nt, uint, uint) {
697         (Error err, uint liquidity, uint shortfal
l) = getHypotheticalAccountLiquidityInternal(accou
nt, CToken(cTokenModify), redeemTokens, borrowAmou
nt);
698         return (uint(err), liquidity, shortfall);
699     }
700     /**
701     * @notice Determine what the account liquidit
y would be if the given amounts were redeemed/borr
owed
702     * @param cTokenModify The market to hypotheti
cally redeem/borrow in
703     * @param account The account to determine liq
uidity for
704     * @param redeemTokens The number of tokens to
hypothetically redeem
705     * @param borrowAmount The amount of underlyin
g to hypothetically borrow
706     * @dev Note that we calculate the exchangeRat
eStored for each collateral cToken using stored da
ta,
707     * without calculating accumulated interest.
708     * @return (possible error code,
              hypothetical account liquidity in
              excess of collateral requirements,
709     *           hypothetical account shortfall bel
ow collateral requirements)
710     */
711     function getHypotheticalAccountLiquidityIntern
al(
712         address account,
713         CToken cTokenModify,
714         uint redeemTokens,
715         uint borrowAmount) internal view returns
(Error, uint, uint) {
716         AccountLiquidityLocalVars memory vars; //
Holds all our calculation results
717         uint oErr;
718         MathError mErr;
719         // For each asset the account is in
720         CToken[] memory assets = accountAssets[acc
ount];
721         for (uint i = 0; i < assets.length; i++) {
722             CToken asset = assets[i];
723             // Read the balances and exchange rate
724             from the cToken
725             (oErr, vars.cTokenBalance, vars.borrow
Balance, vars.exchangeRateMantissa) = asset.getAcc
ountSnapshot(account);

```

```

712             hypothetical account liquidity in
              excess of collateral requirements,
713     *           hypothetical account shortfall bel
ow collateral requirements)
714     */
715     function getHypotheticalAccountLiquidity(
716         address account,
717         address cTokenModify,
718         uint redeemTokens,
719         uint borrowAmount) public view returns (ui
nt, uint, uint) {
720         (Error err, uint liquidity, uint shortfal
l) = getHypotheticalAccountLiquidityInternal(accou
nt, CToken(cTokenModify), redeemTokens, borrowAmou
nt);
721         return (uint(err), liquidity, shortfall);
722     }
723     /**
724     * @notice Determine what the account liquidit
y would be if the given amounts were redeemed/borr
owed
725     * @param cTokenModify The market to hypotheti
cally redeem/borrow in
726     * @param account The account to determine liq
uidity for
727     * @param redeemTokens The number of tokens to
hypothetically redeem
728     * @param borrowAmount The amount of underlyin
g to hypothetically borrow
729     * @dev Note that we calculate the exchangeRat
eStored for each collateral cToken using stored da
ta,
730     * without calculating accumulated interest.
731     * @return (possible error code,
              hypothetical account liquidity in
              excess of collateral requirements,
732     *           hypothetical account shortfall bel
ow collateral requirements)
733     */
734     function getHypotheticalAccountLiquidityIntern
al(
735         address account,
736         CToken cTokenModify,
737         uint redeemTokens,
738         uint borrowAmount) internal view returns
(Error, uint, uint) {
739         AccountLiquidityLocalVars memory vars; //
Holds all our calculation results
740         uint oErr;
741         MathError mErr;
742         // For each asset the account is in
743         CToken[] memory assets = accountAssets[acc
ount];
744         for (uint i = 0; i < assets.length; i++) {
745             CToken asset = assets[i];
746             // Read the balances and exchange rate
747             from the cToken
748             (oErr, vars.cTokenBalance, vars.borrow
Balance, vars.exchangeRateMantissa) = asset.getAcc
ountSnapshot(account);

```

```

730         if (oErr != 0) { // semi-opaque error
code, we assume NO_ERROR == 0 is invariant between
n upgrades
731             return (Error.SNAPSHOT_ERROR, 0,
0);
732         }
733         vars.collateralFactor = Exp({mantissa:
markets[address(asset)].collateralFactorMantiss
a});
734         vars.exchangeRate = Exp({mantissa: var
s.exchangeRateMantissa});
735
736         // Get the normalized price of the ass
et
737         vars.oraclePriceMantissa = oracle.getU
nderlyingPrice(asset);
738         if (vars.oraclePriceMantissa == 0) {
739             return (Error.PRICE_ERROR, 0, 0);
740         }
741         vars.oraclePrice = Exp({mantissa: var
s.oraclePriceMantissa});
742
743         // Pre-compute a conversion factor fro
m tokens -> ether (normalized price value)
744         (mErr, vars.tokensToDenom) = mulExp3(v
ars.collateralFactor, vars.exchangeRate, vars.orac
lePrice);
745         if (mErr != MathError.NO_ERROR) {
746             return (Error.MATH_ERROR, 0, 0);
747         }
748
749         // sumCollateral += tokensToDenom * cT
okenBalance
750         (mErr, vars.sumCollateral) = mulScalar
TruncateAddUInt(vars.tokensToDenom, vars.cTokenBal
ance, vars.sumCollateral);
751         if (mErr != MathError.NO_ERROR) {
752             return (Error.MATH_ERROR, 0, 0);
753         }
754
755         // sumBorrowPlusEffects += oraclePrice
* borrowBalance
756         (mErr, vars.sumBorrowPlusEffects) = mu
lScalarTruncateAddUInt(vars.oraclePrice, vars.borr
owBalance, vars.sumBorrowPlusEffects);
757         if (mErr != MathError.NO_ERROR) {
758             return (Error.MATH_ERROR, 0, 0);
759         }
760
761         // Calculate effects of interacting wi
th cTokenModify
762         if (asset == cTokenModify) {
763             // redeem effect
764             // sumBorrowPlusEffects += tokensT
oDenom * redeemTokens
765             (mErr, vars.sumBorrowPlusEffects)
= mulScalarTruncateAddUInt(vars.tokensToDenom, re
deemTokens, vars.sumBorrowPlusEffects);
766             if (mErr != MathError.NO_ERROR) {
767                 return (Error.MATH_ERROR, 0,
0);
768             }
769
770             // borrow effect
771             // sumBorrowPlusEffects += oracleP
rice * borrowAmount

```

```

753         if (oErr != 0) { // semi-opaque error
code, we assume NO_ERROR == 0 is invariant between
n upgrades
754             return (Error.SNAPSHOT_ERROR, 0,
0);
755         }
756         vars.collateralFactor = Exp({mantissa:
markets[address(asset)].collateralFactorMantiss
a});
757         vars.exchangeRate = Exp({mantissa: var
s.exchangeRateMantissa});
758
759         // Get the normalized price of the ass
et
760         vars.oraclePriceMantissa = oracle.getU
nderlyingPrice(asset);
761         if (vars.oraclePriceMantissa == 0) {
762             return (Error.PRICE_ERROR, 0, 0);
763         }
764         vars.oraclePrice = Exp({mantissa: var
s.oraclePriceMantissa});
765
766         // Pre-compute a conversion factor fro
m tokens -> ether (normalized price value)
767         (mErr, vars.tokensToDenom) = mulExp3(v
ars.collateralFactor, vars.exchangeRate, vars.orac
lePrice);
768         if (mErr != MathError.NO_ERROR) {
769             return (Error.MATH_ERROR, 0, 0);
770         }
771
772         // sumCollateral += tokensToDenom * cT
okenBalance
773         (mErr, vars.sumCollateral) = mulScalar
TruncateAddUInt(vars.tokensToDenom, vars.cTokenBal
ance, vars.sumCollateral);
774         if (mErr != MathError.NO_ERROR) {
775             return (Error.MATH_ERROR, 0, 0);
776         }
777
778         // sumBorrowPlusEffects += oraclePrice
* borrowBalance
779         (mErr, vars.sumBorrowPlusEffects) = mu
lScalarTruncateAddUInt(vars.oraclePrice, vars.borr
owBalance, vars.sumBorrowPlusEffects);
780         if (mErr != MathError.NO_ERROR) {
781             return (Error.MATH_ERROR, 0, 0);
782         }
783
784         // Calculate effects of interacting wi
th cTokenModify
785         if (asset == cTokenModify) {
786             // redeem effect
787             // sumBorrowPlusEffects += tokensT
oDenom * redeemTokens
788             (mErr, vars.sumBorrowPlusEffects)
= mulScalarTruncateAddUInt(vars.tokensToDenom, re
deemTokens, vars.sumBorrowPlusEffects);
789             if (mErr != MathError.NO_ERROR) {
790                 return (Error.MATH_ERROR, 0,
0);
791             }
792
793             // borrow effect
794             // sumBorrowPlusEffects += oracleP
rice * borrowAmount

```

```

772         (mErr, vars.sumBorrowPlusEffects)
    = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
773         if (mErr != MathError.NO_ERROR) {
774             return (Error.MATH_ERROR, 0,
775                 0);
776         }
777     }
778 }
779 // These are safe, as the underflow condition is checked first
780 if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
781     return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
782 } else {
783     return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
784 }
785 }
786 /**
787  * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
788  * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
789  * @param cTokenBorrowed The address of the borrowed cToken
790  * @param cTokenCollateral The address of the collateral cToken
791  * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral tokens
792  * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
793  */
794 function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint actualRepayAmount) external view returns (uint, uint) {
795     /* Read oracle prices for borrowed and collateral markets */
796     uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
797     uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
798     if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
799         return (uint(Error.PRICE_ERROR), 0);
800     }
801 }
802 /**
803  * Get the exchange rate and calculate the number of collateral tokens to seize:
804  * seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
805  * seizeTokens = seizeAmount / exchangeRate
806  * = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
807  */
808 uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on error

```

```

795         (mErr, vars.sumBorrowPlusEffects)
    = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects);
796         if (mErr != MathError.NO_ERROR) {
797             return (Error.MATH_ERROR, 0,
798                 0);
799         }
800     }
801 }
802 // These are safe, as the underflow condition is checked first
803 if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
804     return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
805 } else {
806     return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
807 }
808 }
809 /**
810  * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
811  * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
812  * @param cTokenBorrowed The address of the borrowed cToken
813  * @param cTokenCollateral The address of the collateral cToken
814  * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral tokens
815  * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
816  */
817 function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint actualRepayAmount) external view returns (uint, uint) {
818     /* Read oracle prices for borrowed and collateral markets */
819     uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
820     uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
821     if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
822         return (uint(Error.PRICE_ERROR), 0);
823     }
824 }
825 /**
826  * Get the exchange rate and calculate the number of collateral tokens to seize:
827  * seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
828  * seizeTokens = seizeAmount / exchangeRate
829  * = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
830  */
831 uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on error

```

```

810     uint seizeTokens;
811     Exp memory numerator;
812     Exp memory denominator;
813     Exp memory ratio;
814     MathError mathErr;
815
816     (mathErr, numerator) = mulExp(liquidationI
ncentiveMantissa, priceBorrowedMantissa);
817     if (mathErr != MathError.NO_ERROR) {
818         return (uint(Error.MATH_ERROR), 0);
819     }
820
821     (mathErr, denominator) = mulExp(priceColla
teralMantissa, exchangeRateMantissa);
822     if (mathErr != MathError.NO_ERROR) {
823         return (uint(Error.MATH_ERROR), 0);
824     }
825
826     (mathErr, ratio) = divExp(numerator, denom
inator);
827     if (mathErr != MathError.NO_ERROR) {
828         return (uint(Error.MATH_ERROR), 0);
829     }
830
831     (mathErr, seizeTokens) = mulScalarTruncate
(ratio, actualRepayAmount);
832     if (mathErr != MathError.NO_ERROR) {
833         return (uint(Error.MATH_ERROR), 0);
834     }
835
836     return (uint(Error.NO_ERROR), seizeToken
s);
837 }
838
839 /** Admin Functions */
840
841 /**
842  * @notice Sets a new price oracle for the co
mptroller
843  * @dev Admin function to set a new price ora
cle
844  * @return uint 0=success, otherwise a failur
e (see ErrorReporter.sol for details)
845  */
846 function _setPriceOracle(PriceOracle newOracl
e) public returns (uint) {
847     // Check caller is admin
848     if (msg.sender != admin) {
849         return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_PRICE_ORACLE_OWNER_CHECK);
850     }
851
852     // Track the old oracle for the comptrolle
r
853     PriceOracle oldOracle = oracle;
854
855     // Set comptroller's oracle to newOracle
856     oracle = newOracle;
857
858     // Emit NewPriceOracle(oldOracle, newOracl
e)
859     emit NewPriceOracle(oldOracle, newOracle);
860
861     return uint(Error.NO_ERROR);
862 }
863
864 /**

```

```

833     uint seizeTokens;
834     Exp memory numerator;
835     Exp memory denominator;
836     Exp memory ratio;
837     MathError mathErr;
838
839     (mathErr, numerator) = mulExp(liquidationI
ncentiveMantissa, priceBorrowedMantissa);
840     if (mathErr != MathError.NO_ERROR) {
841         return (uint(Error.MATH_ERROR), 0);
842     }
843
844     (mathErr, denominator) = mulExp(priceColla
teralMantissa, exchangeRateMantissa);
845     if (mathErr != MathError.NO_ERROR) {
846         return (uint(Error.MATH_ERROR), 0);
847     }
848
849     (mathErr, ratio) = divExp(numerator, denom
inator);
850     if (mathErr != MathError.NO_ERROR) {
851         return (uint(Error.MATH_ERROR), 0);
852     }
853
854     (mathErr, seizeTokens) = mulScalarTruncate
(ratio, actualRepayAmount);
855     if (mathErr != MathError.NO_ERROR) {
856         return (uint(Error.MATH_ERROR), 0);
857     }
858
859     return (uint(Error.NO_ERROR), seizeToken
s);
860 }
861
862 /** Admin Functions */
863
864 /**
865  * @notice Sets a new price oracle for the co
mptroller
866  * @dev Admin function to set a new price ora
cle
867  * @return uint 0=success, otherwise a failur
e (see ErrorReporter.sol for details)
868  */
869 function _setPriceOracle(PriceOracle newOracl
e) public returns (uint) {
870     // Check caller is admin
871     if (msg.sender != admin) {
872         return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_PRICE_ORACLE_OWNER_CHECK);
873     }
874
875     // Track the old oracle for the comptrolle
r
876     PriceOracle oldOracle = oracle;
877
878     // Set comptroller's oracle to newOracle
879     oracle = newOracle;
880
881     // Emit NewPriceOracle(oldOracle, newOracl
e)
882     emit NewPriceOracle(oldOracle, newOracle);
883
884     return uint(Error.NO_ERROR);
885 }
886
887 /**

```

```

865     * @notice Sets the closeFactor used when liq
uidating borrows
866     * @dev Admin function to set closeFactor
867     * @param newCloseFactorMantissa New close fa
ctor, scaled by 1e18
868     * @return uint 0=success, otherwise a failur
e. (See ErrorReporter for details)
869     */
870     function _setCloseFactor(uint newCloseFactorMa
ntissa) external returns (uint) {
871         // Check caller is admin
872         if (msg.sender != admin) {
873             return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
874         }
875
876         Exp memory newCloseFactorExp = Exp({mantis
sa: newCloseFactorMantissa});
877         Exp memory lowLimit = Exp({mantissa: close
FactorMinMantissa});
878         if (lessThanOrEqualExp(newCloseFactorExp,
lowLimit)) {
879             return fail(Error.INVALID_CLOSE_FACTO
R, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
880         }
881
882         Exp memory highLimit = Exp({mantissa: clos
eFactorMaxMantissa});
883         if (lessThanExp(highLimit, newCloseFactorE
xp)) {
884             return fail(Error.INVALID_CLOSE_FACTO
R, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
885         }
886
887         uint oldCloseFactorMantissa = closeFactorM
antissa;
888         closeFactorMantissa = newCloseFactorMantis
sa;
889         emit NewCloseFactor(oldCloseFactorMantiss
a, closeFactorMantissa);
890
891         return uint(Error.NO_ERROR);
892     }
893
894     /**
895     * @notice Sets the collateralFactor for a ma
rket
896     * @dev Admin function to set per-market coll
ateralFactor
897     * @param cToken The market to set the factor
on
898     * @param newCollateralFactorMantissa The new
collateral factor, scaled by 1e18
899     * @return uint 0=success, otherwise a failur
e. (See ErrorReporter for details)
900     */
901     function _setCollateralFactor(CToken cToken, u
int newCollateralFactorMantissa) external returns
(uint) {
902         // Check caller is admin
903         if (msg.sender != admin) {
904             return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
905         }
906
907         // Verify market is listed

```

```

888     * @notice Sets the closeFactor used when liq
uidating borrows
889     * @dev Admin function to set closeFactor
890     * @param newCloseFactorMantissa New close fa
ctor, scaled by 1e18
891     * @return uint 0=success, otherwise a failur
e. (See ErrorReporter for details)
892     */
893     function _setCloseFactor(uint newCloseFactorMa
ntissa) external returns (uint) {
894         // Check caller is admin
895         if (msg.sender != admin) {
896             return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
897         }
898
899         Exp memory newCloseFactorExp = Exp({mantis
sa: newCloseFactorMantissa});
900         Exp memory lowLimit = Exp({mantissa: close
FactorMinMantissa});
901         if (lessThanOrEqualExp(newCloseFactorExp,
lowLimit)) {
902             return fail(Error.INVALID_CLOSE_FACTO
R, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
903         }
904
905         Exp memory highLimit = Exp({mantissa: clos
eFactorMaxMantissa});
906         if (lessThanExp(highLimit, newCloseFactorE
xp)) {
907             return fail(Error.INVALID_CLOSE_FACTO
R, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
908         }
909
910         uint oldCloseFactorMantissa = closeFactorM
antissa;
911         closeFactorMantissa = newCloseFactorMantis
sa;
912         emit NewCloseFactor(oldCloseFactorMantiss
a, closeFactorMantissa);
913
914         return uint(Error.NO_ERROR);
915     }
916
917     /**
918     * @notice Sets the collateralFactor for a ma
rket
919     * @dev Admin function to set per-market coll
ateralFactor
920     * @param cToken The market to set the factor
on
921     * @param newCollateralFactorMantissa The new
collateral factor, scaled by 1e18
922     * @return uint 0=success, otherwise a failur
e. (See ErrorReporter for details)
923     */
924     function _setCollateralFactor(CToken cToken, u
int newCollateralFactorMantissa) external returns
(uint) {
925         // Check caller is admin
926         if (msg.sender != admin) {
927             return fail(Error.UNAUTHORIZED, Failur
eInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
928         }
929
930         // Verify market is listed

```

```

908         Market storage market = markets[address(cToken)];
909         if (!market.isListed) {
910             return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
911         }
912
913         Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});
914
915         // Check collateral factor <= 0.9
916         Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
917         if (lessThanExp(highLimit, newCollateralFactorExp)) {
918             return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
919         }
920
921         // If collateral factor != 0, fail if price == 0
922         if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
923             return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
924         }
925
926         // Set market's collateral factor to new collateral factor, remember old value
927         uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
928         market.collateralFactorMantissa = newCollateralFactorMantissa;
929
930         // Emit event with asset, old collateral factor, and new collateral factor
931         emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);
932
933         return uint(Error.NO_ERROR);
934     }
935
936     /**
937      * @notice Sets maxAssets which controls how many markets can be entered
938      * @dev Admin function to set maxAssets
939      * @param newMaxAssets New max assets
940      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
941      */
942     function _setMaxAssets(uint newMaxAssets) external returns (uint) {
943         // Check caller is admin
944         if (msg.sender != admin) {
945             return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
946         }
947
948         uint oldMaxAssets = maxAssets;
949         maxAssets = newMaxAssets;
950         emit NewMaxAssets(oldMaxAssets, newMaxAssets);
951
952         return uint(Error.NO_ERROR);
953     }
954

```

```

931         Market storage market = markets[address(cToken)];
932         if (!market.isListed) {
933             return fail(Error.MARKET_NOT_LISTED, FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
934         }
935
936         Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});
937
938         // Check collateral factor <= 0.9
939         Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
940         if (lessThanExp(highLimit, newCollateralFactorExp)) {
941             return fail(Error.INVALID_COLLATERAL_FACTOR, FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
942         }
943
944         // If collateral factor != 0, fail if price == 0
945         if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
946             return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
947         }
948
949         // Set market's collateral factor to new collateral factor, remember old value
950         uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
951         market.collateralFactorMantissa = newCollateralFactorMantissa;
952
953         // Emit event with asset, old collateral factor, and new collateral factor
954         emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);
955
956         return uint(Error.NO_ERROR);
957     }
958
959     /**
960      * @notice Sets maxAssets which controls how many markets can be entered
961      * @dev Admin function to set maxAssets
962      * @param newMaxAssets New max assets
963      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
964      */
965     function _setMaxAssets(uint newMaxAssets) external returns (uint) {
966         // Check caller is admin
967         if (msg.sender != admin) {
968             return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
969         }
970
971         uint oldMaxAssets = maxAssets;
972         maxAssets = newMaxAssets;
973         emit NewMaxAssets(oldMaxAssets, newMaxAssets);
974
975         return uint(Error.NO_ERROR);
976     }
977

```

```

955     /**
956     * @notice Sets liquidationIncentive
957     * @dev Admin function to set liquidationIncentive
958     * @param newLiquidationIncentiveMantissa New
959     * liquidationIncentive scaled by 1e18
960     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
961     */
962     function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint)
963     {
964         // Check caller is admin
965         if (msg.sender != admin) {
966             return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
967         }
968         // Check de-scaled min <= newLiquidationIncentive <= max
969         Exp memory newLiquidationIncentive = Exp
970         ({mantissa: newLiquidationIncentiveMantissa});
971         Exp memory minLiquidationIncentive = Exp
972         ({mantissa: liquidationIncentiveMinMantissa});
973         if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
974             return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
975         }
976         Exp memory maxLiquidationIncentive = Exp
977         ({mantissa: liquidationIncentiveMaxMantissa});
978         if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
979             return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
980         }
981         // Save current value for use in log
982         uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;
983         // Set liquidation incentive to new incentive
984         liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;
985         // Emit event with old incentive, new incentive
986         emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);
987         return uint(Error.NO_ERROR);
988     }
989     /**
990     * @notice Add the market to the markets mapping and set it as listed
991     * @dev Admin function to set isListed and add support for the market
992     * @param cToken The address of the market (token) to list
993     * @return uint 0=success, otherwise a failure. (See enum Error for details)

```

```

978     /**
979     * @notice Sets liquidationIncentive
980     * @dev Admin function to set liquidationIncentive
981     * @param newLiquidationIncentiveMantissa New
982     * liquidationIncentive scaled by 1e18
983     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
984     */
985     function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint)
986     {
987         // Check caller is admin
988         if (msg.sender != admin) {
989             return fail(Error.UNAUTHORIZED, FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
990         }
991         // Check de-scaled min <= newLiquidationIncentive <= max
992         Exp memory newLiquidationIncentive = Exp
993         ({mantissa: newLiquidationIncentiveMantissa});
994         Exp memory minLiquidationIncentive = Exp
995         ({mantissa: liquidationIncentiveMinMantissa});
996         if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
997             return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
998         }
999         Exp memory maxLiquidationIncentive = Exp
1000         ({mantissa: liquidationIncentiveMaxMantissa});
1001         if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
1002             return fail(Error.INVALID_LIQUIDATION_INCENTIVE, FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
1003         }
1004         // Save current value for use in log
1005         uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;
1006         // Set liquidation incentive to new incentive
1007         liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;
1008         // Emit event with old incentive, new incentive
1009         emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);
1010         return uint(Error.NO_ERROR);
1011     }
1012     /**
1013     * @notice Add the market to the markets mapping and set it as listed
1014     * @dev Admin function to set isListed and add support for the market
1015     * @param cToken The address of the market (token) to list
1016     * @return uint 0=success, otherwise a failure. (See enum Error for details)

```



```

996     */
997     function _supportMarket(CToken cToken) external returns (uint) {
998         if (msg.sender != admin) {
999             return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
1000         }
1001
1002         if (markets[address(cToken)].isListed) {
1003             return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
1004         }
1005
1006         cToken.isCToken(); // Sanity check to make sure its really a CToken
1007
1008         markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});
1009
1010         _addMarketInternal(address(cToken));
1011
1012         emit MarketListed(cToken);
1013
1014         return uint(Error.NO_ERROR);
1015     }
1016
1017     function _addMarketInternal(address cToken) internal {
1018         for (uint i = 0; i < allMarkets.length; i++) {
1019             require(allMarkets[i] != CToken(cToken), "market already added");
1020         }
1021         allMarkets.push(CToken(cToken));
1022     }
1023
1024     /**
1025      * @notice Admin function to change the Pause Guardian
1026      * @param newPauseGuardian The address of the new Pause Guardian
1027      * @return uint 0=success, otherwise a failure. (See enum Error for details)
1028      */
1029     function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
1030         if (msg.sender != admin) {
1031             return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
1032         }
1033
1034         // Save current value for inclusion in log
1035         address oldPauseGuardian = pauseGuardian;
1036
1037         // Store pauseGuardian with value newPauseGuardian
1038         pauseGuardian = newPauseGuardian;
1039
1040         // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
1041         emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);
1042
1043         return uint(Error.NO_ERROR);
1044     }
1045

```

```

1019     */
1020     function _supportMarket(CToken cToken) external returns (uint) {
1021         if (msg.sender != admin) {
1022             return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
1023         }
1024
1025         if (markets[address(cToken)].isListed) {
1026             return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
1027         }
1028
1029         cToken.isCToken(); // Sanity check to make sure its really a CToken
1030
1031         markets[address(cToken)] = Market({isListed: true, isComped: false, collateralFactorMantissa: 0});
1032
1033         _addMarketInternal(address(cToken));
1034
1035         emit MarketListed(cToken);
1036
1037         return uint(Error.NO_ERROR);
1038     }
1039
1040     function _addMarketInternal(address cToken) internal {
1041         for (uint i = 0; i < allMarkets.length; i++) {
1042             require(allMarkets[i] != CToken(cToken), "market already added");
1043         }
1044         allMarkets.push(CToken(cToken));
1045     }
1046
1047     /**
1048      * @notice Admin function to change the Pause Guardian
1049      * @param newPauseGuardian The address of the new Pause Guardian
1050      * @return uint 0=success, otherwise a failure. (See enum Error for details)
1051      */
1052     function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
1053         if (msg.sender != admin) {
1054             return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
1055         }
1056
1057         // Save current value for inclusion in log
1058         address oldPauseGuardian = pauseGuardian;
1059
1060         // Store pauseGuardian with value newPauseGuardian
1061         pauseGuardian = newPauseGuardian;
1062
1063         // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
1064         emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);
1065
1066         return uint(Error.NO_ERROR);
1067     }
1068

```

```

1046     function _setMintPaused(CToken cToken, bool st
ate) public returns (bool) {
1047         require(markets[address(cToken)].isListed,
"cannot pause a market that is not listed");
1048         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1049         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1050
1051         mintGuardianPaused[address(cToken)] = stat
e;
1052         emit ActionPaused(cToken, "Mint", state);
1053         return state;
1054     }
1055
1056     function _setBorrowPaused(CToken cToken, bool
state) public returns (bool) {
1057         require(markets[address(cToken)].isListed,
"cannot pause a market that is not listed");
1058         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1059         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1060
1061         borrowGuardianPaused[address(cToken)] = st
ate;
1062         emit ActionPaused(cToken, "Borrow", stat
e);
1063         return state;
1064     }
1065
1066     function _setTransferPaused(bool state) public
returns (bool) {
1067         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1068         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1069
1070         transferGuardianPaused = state;
1071         emit ActionPaused("Transfer", state);
1072         return state;
1073     }
1074
1075     function _setSeizePaused(bool state) public re
turns (bool) {
1076         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1077         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1078
1079         seizeGuardianPaused = state;
1080         emit ActionPaused("Seize", state);
1081         return state;
1082     }
1083
1084     function _become(Unitroller unitroller) public
{
1085         require(msg.sender == unitroller.admin(),
"only unitroller admin can change brains");
1086         require(unitroller._acceptImplementation()
== 0, "change not authorized");
1087     }
1088

```

```

1069     function _setMintPaused(CToken cToken, bool st
ate) public returns (bool) {
1070         require(markets[address(cToken)].isListed,
"cannot pause a market that is not listed");
1071         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1072         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1073
1074         mintGuardianPaused[address(cToken)] = stat
e;
1075         emit ActionPaused(cToken, "Mint", state);
1076         return state;
1077     }
1078
1079     function _setBorrowPaused(CToken cToken, bool
state) public returns (bool) {
1080         require(markets[address(cToken)].isListed,
"cannot pause a market that is not listed");
1081         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1082         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1083
1084         borrowGuardianPaused[address(cToken)] = st
ate;
1085         emit ActionPaused(cToken, "Borrow", stat
e);
1086         return state;
1087     }
1088
1089     function _setTransferPaused(bool state) public
returns (bool) {
1090         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1091         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1092
1093         transferGuardianPaused = state;
1094         emit ActionPaused("Transfer", state);
1095         return state;
1096     }
1097
1098     function _setSeizePaused(bool state) public re
turns (bool) {
1099         require(msg.sender == pauseGuardian || ms
g.sender == admin, "only pause guardian and admin
can pause");
1100         require(msg.sender == admin || state == tr
ue, "only admin can unpause");
1101
1102         seizeGuardianPaused = state;
1103         emit ActionPaused("Seize", state);
1104         return state;
1105     }
1106
1107     function _become(Unitroller unitroller) public
{
1108         require(msg.sender == unitroller.admin(),
"only unitroller admin can change brains");
1109         require(unitroller._acceptImplementation()
== 0, "change not authorized");
1110     }
1111

```

```

1089     /**
1090      * @notice Checks caller is admin, or this con
      tract is becoming the new implementation
1091      */
1092      function adminOrInitializing() internal view r
      eturns (bool) {
1093          return msg.sender == admin || msg.sender =
      = comptrollerImplementation;
1094      }
1095
1096      /** Comp Distribution */
1097
1098      /**
1099       * @notice Recalculate and update COMP speeds
      for all COMP markets
1100       */
1101      function refreshCompSpeeds() public {
1102          require(msg.sender == tx.origin, "only ext
      ernally owned accounts may refresh speeds");
1103          refreshCompSpeedsInternal();
1104      }
1105
1106      function refreshCompSpeedsInternal() internal
      {
1107          CToken[] memory allMarkets_ = allMarkets;
1108
1109          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1110              CToken cToken = allMarkets_[i];
1111              Exp memory borrowIndex = Exp({mantiss
      a: cToken.borrowIndex()});
1112              updateCompSupplyIndex(address(cToke
      n));
1113              updateCompBorrowIndex(address(cToken),
      borrowIndex);
1114          }
1115
1116          Exp memory totalUtility = Exp({mantissa:
      0});
1117          Exp[] memory utilities = new Exp[](allMark
      ets_.length);
1118          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1119              CToken cToken = allMarkets_[i];
1120              if (markets[address(cToken)].isComped)
      {
1121                  Exp memory assetPrice = Exp({manti
      ssa: oracle.getUnderlyingPrice(cToken)});
1122                  Exp memory utility = mul_(assetPri
      ce, cToken.totalBorrows());
1123                  utilities[i] = utility;
1124                  totalUtility = add_(totalUtility,
      utility);
1125              }
1126          }
1127
1128          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1129              CToken cToken = allMarkets[i];
1130              uint newSpeed = totalUtility.mantissa
      > 0 ? mul_(compRate, div_(utilities[i], totalUtil
      ity)) : 0;
1131              compSpeeds[address(cToken)] = newSpee
      d;

```

```

1112     /**
1113      * @notice Checks caller is admin, or this con
      tract is becoming the new implementation
1114      */
1115      function adminOrInitializing() internal view r
      eturns (bool) {
1116          return msg.sender == admin || msg.sender =
      = comptrollerImplementation;
1117      }
1118
1119      /** Comp Distribution */
1120
1121      /**
1122       * @notice Recalculate and update COMP speeds
      for all COMP markets
1123       */
1124      function refreshCompSpeeds() public {
1125          require(msg.sender == tx.origin, "only ext
      ernally owned accounts may refresh speeds");
1126          refreshCompSpeedsInternal();
1127      }
1128
1129      function refreshCompSpeedsInternal() internal
      {
1130          CToken[] memory allMarkets_ = allMarkets;
1131
1132          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1133              CToken cToken = allMarkets_[i];
1134              Exp memory borrowIndex = Exp({mantiss
      a: cToken.borrowIndex()});
1135              updateCompSupplyIndex(address(cToke
      n));
1136              updateCompBorrowIndex(address(cToken),
      borrowIndex);
1137          }
1138
1139          Exp memory totalUtility = Exp({mantissa:
      0});
1140          Exp[] memory utilities = new Exp[](allMark
      ets_.length);
1141          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1142              CToken cToken = allMarkets_[i];
1143              if (markets[address(cToken)].isComped)
      {
1144                  oracle.updatePrice(cToken);
1145                  Exp memory assetPrice = Exp({manti
      ssa: oracle.getUnderlyingPrice(cToken)});
1146                  Exp memory utility = mul_(assetPri
      ce, cToken.totalBorrows());
1147                  utilities[i] = utility;
1148                  totalUtility = add_(totalUtility,
      utility);
1149              }
1150          }
1151
1152          for (uint i = 0; i < allMarkets_.length; i
      ++ ) {
1153              CToken cToken = allMarkets[i];
1154              uint newSpeed = totalUtility.mantissa
      > 0 ? mul_(compRate, div_(utilities[i], totalUtil
      ity)) : 0;
1155              compSpeeds[address(cToken)] = newSpee
      d;

```

```

1132         emit CompSpeedUpdated(cToken, newSpee
1133 d);
1134     }
1135 }
1136 /**
1137  * @notice Accrue COMP to the market by updati
1138 ng the supply index
1139  * @param cToken The market whose supply index
1140 to update
1141 */
1142 function updateCompSupplyIndex(address cToken)
1143 internal {
1144     CompMarketState storage supplyState = comp
1145 SupplyState[cToken];
1146     uint supplySpeed = compSpeeds[cToken];
1147     uint blockNumber = getBlockNumber();
1148     uint deltaBlocks = sub_(blockNumber, uint
1149 (supplyState.block));
1150     if (deltaBlocks > 0 && supplySpeed > 0) {
1151         uint supplyTokens = CToken(cToken).tot
1152 alSupply();
1153         uint compAccrued = mul_(deltaBlocks, s
1154 upplySpeed);
1155         Double memory ratio = supplyTokens > 0
1156 ? fraction(compAccrued, supplyTokens) : Double({ma
1157 ntissa: 0});
1158         Double memory index = add_(Double({man
1159 tissa: supplyState.index}), ratio);
1160         compSupplyState[cToken] = CompMarketSt
1161 ate({
1162             index: safe224(index.mantissa, "ne
1163 w index exceeds 224 bits"),
1164             block: safe32(blockNumber, "block
1165 number exceeds 32 bits")
1166         });
1167     } else if (deltaBlocks > 0) {
1168         supplyState.block = safe32(blockNumbe
1169 r, "block number exceeds 32 bits");
1170     }
1171 }
1172 /**
1173  * @notice Accrue COMP to the market by updati
1174 ng the borrow index
1175  * @param cToken The market whose borrow index
1176 to update
1177 */
1178 function updateCompBorrowIndex(address cToken,
1179 Exp memory marketBorrowIndex) internal {
1180     CompMarketState storage borrowState = comp
1181 BorrowState[cToken];
1182     uint borrowSpeed = compSpeeds[cToken];
1183     uint blockNumber = getBlockNumber();
1184     uint deltaBlocks = sub_(blockNumber, uint
1185 (borrowState.block));
1186     if (deltaBlocks > 0 && borrowSpeed > 0) {
1187         uint borrowAmount = div_(CToken(cToke
1188 n).totalBorrows(), marketBorrowIndex);
1189         uint compAccrued = mul_(deltaBlocks, b
1190 orrowSpeed);
1191         Double memory ratio = borrowAmount > 0
1192 ? fraction(compAccrued, borrowAmount) : Double({ma
1193 ntissa: 0});

```

```

1156         emit CompSpeedUpdated(cToken, newSpee
1157 d);
1158     }
1159 }
1160 /**
1161  * @notice Accrue COMP to the market by updati
1162 ng the supply index
1163  * @param cToken The market whose supply index
1164 to update
1165 */
1166 function updateCompSupplyIndex(address cToken)
1167 internal {
1168     CompMarketState storage supplyState = comp
1169 SupplyState[cToken];
1170     uint supplySpeed = compSpeeds[cToken];
1171     uint blockNumber = getBlockNumber();
1172     uint deltaBlocks = sub_(blockNumber, uint
1173 (supplyState.block));
1174     if (deltaBlocks > 0 && supplySpeed > 0) {
1175         uint supplyTokens = CToken(cToken).tot
1176 alSupply();
1177         uint compAccrued = mul_(deltaBlocks, s
1178 upplySpeed);
1179         Double memory ratio = supplyTokens > 0
1180 ? fraction(compAccrued, supplyTokens) : Double({ma
1181 ntissa: 0});
1182         Double memory index = add_(Double({man
1183 tissa: supplyState.index}), ratio);
1184         compSupplyState[cToken] = CompMarketSt
1185 ate({
1186             index: safe224(index.mantissa, "ne
1187 w index exceeds 224 bits"),
1188             block: safe32(blockNumber, "block
1189 number exceeds 32 bits")
1190         });
1191     } else if (deltaBlocks > 0) {
1192         supplyState.block = safe32(blockNumbe
1193 r, "block number exceeds 32 bits");
1194     }
1195 }
1196 /**
1197  * @notice Accrue COMP to the market by updati
1198 ng the borrow index
1199  * @param cToken The market whose borrow index
1200 to update
1201 */
1202 function updateCompBorrowIndex(address cToken,
1203 Exp memory marketBorrowIndex) internal {
1204     CompMarketState storage borrowState = comp
1205 BorrowState[cToken];
1206     uint borrowSpeed = compSpeeds[cToken];
1207     uint blockNumber = getBlockNumber();
1208     uint deltaBlocks = sub_(blockNumber, uint
1209 (borrowState.block));
1210     if (deltaBlocks > 0 && borrowSpeed > 0) {
1211         uint borrowAmount = div_(CToken(cToke
1212 n).totalBorrows(), marketBorrowIndex);
1213         uint compAccrued = mul_(deltaBlocks, b
1214 orrowSpeed);
1215         Double memory ratio = borrowAmount > 0
1216 ? fraction(compAccrued, borrowAmount) : Double({ma
1217 ntissa: 0});

```

```

1172         Double memory index = add_(Double({man
tissa: borrowState.index}), ratio);
1173         compBorrowState[cToken] = CompMarketSt
ate({
1174             index: safe224(index.mantissa, "ne
w index exceeds 224 bits"),
1175             block: safe32(blockNumber, "block
number exceeds 32 bits")
1176         });
1177     } else if (deltaBlocks > 0) {
1178         borrowState.block = safe32(blockNumbe
r, "block number exceeds 32 bits");
1179     }
1180 }
1181 /**
1182  * @notice Calculate COMP accrued by a supplie
r and possibly transfer it to them
1183  * @param cToken The market in which the suppl
ier is interacting
1184  * @param supplier The address of the supplier
to distribute COMP to
1185  */
1186 function distributeSupplierComp(address cToke
n, address supplier, bool distributeAll) internal
{
1187     CompMarketState storage supplyState = comp
SupplyState[cToken];
1188     Double memory supplyIndex = Double({mantis
sa: supplyState.index});
1189     Double memory supplierIndex = Double({mant
issa: compSupplierIndex[cToken][supplier]});
1190     compSupplierIndex[cToken][supplier] = supp
lyIndex.mantissa;
1191     if (supplierIndex.mantissa == 0 && supplyI
ndex.mantissa > 0) {
1192         supplierIndex.mantissa = compInitialIn
dex;
1193     }
1194     Double memory deltaIndex = sub_(supplyInde
x, supplierIndex);
1195     uint supplierTokens = CToken(cToken).balan
ceOf(supplier);
1196     uint supplierDelta = mul_(supplierTokens,
deltaIndex);
1197     uint supplierAccrued = add_(compAccrued[su
pplier], supplierDelta);
1198     compAccrued[supplier] = transferComp(suppl
ier, supplierAccrued, distributeAll ? 0 : compClai
mThreshold);
1199     emit DistributedSupplierComp(CToken(cToke
n), supplier, supplierDelta, supplyIndex.mantiss
a);
1200 }
1201 /**
1202  * @notice Calculate COMP accrued by a borrowe
r and possibly transfer it to them
1203  * @dev Borrowers will not begin to accrue unt
il after the first interaction with the protocol.
1204  * @param cToken The market in which the borro
wer is interacting
1205  * @param borrower The address of the borrower
to distribute COMP to

```

```

1196         Double memory index = add_(Double({man
tissa: borrowState.index}), ratio);
1197         compBorrowState[cToken] = CompMarketSt
ate({
1198             index: safe224(index.mantissa, "ne
w index exceeds 224 bits"),
1199             block: safe32(blockNumber, "block
number exceeds 32 bits")
1200         });
1201     } else if (deltaBlocks > 0) {
1202         borrowState.block = safe32(blockNumbe
r, "block number exceeds 32 bits");
1203     }
1204 }
1205 /**
1206  * @notice Calculate COMP accrued by a supplie
r and possibly transfer it to them
1207  * @param cToken The market in which the suppl
ier is interacting
1208  * @param supplier The address of the supplier
to distribute COMP to
1209  */
1210 function distributeSupplierComp(address cToke
n, address supplier, bool distributeAll) internal
{
1211     CompMarketState storage supplyState = comp
SupplyState[cToken];
1212     Double memory supplyIndex = Double({mantis
sa: supplyState.index});
1213     Double memory supplierIndex = Double({mant
issa: compSupplierIndex[cToken][supplier]});
1214     compSupplierIndex[cToken][supplier] = supp
lyIndex.mantissa;
1215     if (supplierIndex.mantissa == 0 && supplyI
ndex.mantissa > 0) {
1216         supplierIndex.mantissa = compInitialIn
dex;
1217     }
1218     Double memory deltaIndex = sub_(supplyInde
x, supplierIndex);
1219     uint supplierTokens = CToken(cToken).balan
ceOf(supplier);
1220     uint supplierDelta = mul_(supplierTokens,
deltaIndex);
1221     uint supplierAccrued = add_(compAccrued[su
pplier], supplierDelta);
1222     compAccrued[supplier] = transferComp(suppl
ier, supplierAccrued, distributeAll ? 0 : compClai
mThreshold);
1223     emit DistributedSupplierComp(CToken(cToke
n), supplier, supplierDelta, supplyIndex.mantiss
a);
1224 }
1225 /**
1226  * @notice Calculate COMP accrued by a borrowe
r and possibly transfer it to them
1227  * @dev Borrowers will not begin to accrue unt
il after the first interaction with the protocol.
1228  * @param cToken The market in which the borro
wer is interacting
1229  * @param borrower The address of the borrower
to distribute COMP to

```

```

1210     */
1211     function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex,
bool distributeAll) internal {
1212         CompMarketState storage borrowState = comp
BorrowState[cToken];
1213         Double memory borrowIndex = Double({mantis
sa: borrowState.index});
1214         Double memory borrowerIndex = Double({mant
issa: compBorrowerIndex[cToken][borrower]});
1215         compBorrowerIndex[cToken][borrower] = borr
owIndex.mantissa;
1216
1217         if (borrowerIndex.mantissa > 0) {
1218             Double memory deltaIndex = sub_(borrow
Index, borrowerIndex);
1219             uint borrowerAmount = div_(CToken(cTok
en).borrowBalanceStored(borrower), marketBorrowInd
ex);
1220             uint borrowerDelta = mul_(borrowerAmou
nt, deltaIndex);
1221             uint borrowerAccrued = add_(compAccrue
d[borrower], borrowerDelta);
1222             compAccrued[borrower] = transferComp(b
orrower, borrowerAccrued, distributeAll ? 0 : comp
ClaimThreshold);
1223             emit DistributedBorrowerComp(CToken(cT
oken), borrower, borrowerDelta, borrowIndex.mantis
sa);
1224         }
1225     }
1226
1227     /**
1228     * @notice Transfer COMP to the user, if they
are above the threshold
1229     * @dev Note: If there is not enough COMP, we
do not perform the transfer all.
1230     * @param user The address of the user to tran
sfer COMP to
1231     * @param userAccrued The amount of COMP to (p
ossibly) transfer
1232     * @return The amount of COMP which was NOT tr
ansferred to the user
1233     */
1234     function transferComp(address user, uint userA
ccrued, uint threshold) internal returns (uint) {
1235         if (userAccrued >= threshold && userAccrue
d > 0) {
1236             Comp comp = Comp(getCompAddress());
1237             uint compRemaining = comp.balanceOf(ad
dress(this));
1238             if (userAccrued <= compRemaining) {
1239                 comp.transfer(user, userAccrued);
1240
1241                 return 0;
1242             }
1243             return userAccrued;
1244         }
1245
1246         /**
1247         * @notice Claim all the comp accrued by holde
r in all markets
1248         * @param holder The address to claim COMP for
1249         */
1250         function claimComp(address holder) public {

```

```

1234     */
1235     function distributeBorrowerComp(address cToken, address borrower, Exp memory marketBorrowIndex,
bool distributeAll) internal {
1236         CompMarketState storage borrowState = comp
BorrowState[cToken];
1237         Double memory borrowIndex = Double({mantis
sa: borrowState.index});
1238         Double memory borrowerIndex = Double({mant
issa: compBorrowerIndex[cToken][borrower]});
1239         compBorrowerIndex[cToken][borrower] = borr
owIndex.mantissa;
1240
1241         if (borrowerIndex.mantissa > 0) {
1242             Double memory deltaIndex = sub_(borrow
Index, borrowerIndex);
1243             uint borrowerAmount = div_(CToken(cTok
en).borrowBalanceStored(borrower), marketBorrowInd
ex);
1244             uint borrowerDelta = mul_(borrowerAmou
nt, deltaIndex);
1245             uint borrowerAccrued = add_(compAccrue
d[borrower], borrowerDelta);
1246             compAccrued[borrower] = transferComp(b
orrower, borrowerAccrued, distributeAll ? 0 : comp
ClaimThreshold);
1247             emit DistributedBorrowerComp(CToken(cT
oken), borrower, borrowerDelta, borrowIndex.mantis
sa);
1248         }
1249     }
1250
1251     /**
1252     * @notice Transfer COMP to the user, if they
are above the threshold
1253     * @dev Note: If there is not enough COMP, we
do not perform the transfer all.
1254     * @param user The address of the user to tran
sfer COMP to
1255     * @param userAccrued The amount of COMP to (p
ossibly) transfer
1256     * @return The amount of COMP which was NOT tr
ansferred to the user
1257     */
1258     function transferComp(address user, uint userA
ccrued, uint threshold) internal returns (uint) {
1259         // if (userAccrued >= threshold && userAcc
rued > 0) {
1260             //     Comp comp = Comp(getCompAddress());
1261             //     uint compRemaining = comp.balanceOf
(address(this));
1262             //     if (userAccrued <= compRemaining) {
1263             //         comp.transfer(user, userAccrue
d);
1264             //         return 0;
1265             //     }
1266             // }
1267             return userAccrued;
1268         }
1269
1270         /**
1271         * @notice Claim all the comp accrued by holde
r in all markets
1272         * @param holder The address to claim COMP for
1273         */
1274         function claimComp(address holder) public {

```

```

1251         return claimComp(holder, allMarkets);
1252     }
1253
1254     /**
1255      * @notice Claim all the comp accrued by holde
r in the specified markets
1256      * @param holder The address to claim COMP for
1257      * @param cTokens The list of markets to claim
COMP in
1258      */
1259     function claimComp(address holder, CToken[] me
memory cTokens) public {
1260         address[] memory holders = new address[]
(1);
1261         holders[0] = holder;
1262         claimComp(holders, cTokens, true, true);
1263     }
1264
1265     /**
1266      * @notice Claim all comp accrued by the holde
rs
1267      * @param holders The addresses to claim COMP
for
1268      * @param cTokens The list of markets to claim
COMP in
1269      * @param borrowers Whether or not to claim CO
MP earned by borrowing
1270      * @param suppliers Whether or not to claim CO
MP earned by supplying
1271      */
1272     function claimComp(address[] memory holders, C
Token[] memory cTokens, bool borrowers, bool suppl
iers) public {
1273         for (uint i = 0; i < cTokens.length; i++)
        {
1274             CToken cToken = cTokens[i];
1275             require(markets[address(cToken)].isLis
ted, "market must be listed");
1276             if (borrowers == true) {
1277                 Exp memory borrowIndex = Exp({mant
issa: cToken.borrowIndex()});
1278                 updateCompBorrowIndex(address(cTok
en), borrowIndex);
1279                 for (uint j = 0; j < holders.lengt
h; j++) {
1280                     distributeBorrowerComp(address
(cToken), holders[j], borrowIndex, true);
1281                 }
1282             }
1283             if (suppliers == true) {
1284                 updateCompSupplyIndex(address(cTok
en));
1285                 for (uint j = 0; j < holders.lengt
h; j++) {
1286                     distributeSupplierComp(address
(cToken), holders[j], true);
1287                 }
1288             }
1289         }
1290     }
1291
1292     /** Comp Distribution Admin */
1293
1294     /**
1295      * @notice Set the amount of COMP distributed
per block

```

```

1275         return claimComp(holder, allMarkets);
1276     }
1277
1278     /**
1279      * @notice Claim all the comp accrued by holde
r in the specified markets
1280      * @param holder The address to claim COMP for
1281      * @param cTokens The list of markets to claim
COMP in
1282      */
1283     function claimComp(address holder, CToken[] me
memory cTokens) public {
1284         address[] memory holders = new address[]
(1);
1285         holders[0] = holder;
1286         claimComp(holders, cTokens, true, true);
1287     }
1288
1289     /**
1290      * @notice Claim all comp accrued by the holde
rs
1291      * @param holders The addresses to claim COMP
for
1292      * @param cTokens The list of markets to claim
COMP in
1293      * @param borrowers Whether or not to claim CO
MP earned by borrowing
1294      * @param suppliers Whether or not to claim CO
MP earned by supplying
1295      */
1296     function claimComp(address[] memory holders, C
Token[] memory cTokens, bool borrowers, bool suppl
iers) public {
1297         for (uint i = 0; i < cTokens.length; i++)
        {
1298             CToken cToken = cTokens[i];
1299             require(markets[address(cToken)].isLis
ted, "market must be listed");
1300             if (borrowers == true) {
1301                 Exp memory borrowIndex = Exp({mant
issa: cToken.borrowIndex()});
1302                 updateCompBorrowIndex(address(cTok
en), borrowIndex);
1303                 for (uint j = 0; j < holders.lengt
h; j++) {
1304                     distributeBorrowerComp(address
(cToken), holders[j], borrowIndex, true);
1305                 }
1306             }
1307             if (suppliers == true) {
1308                 updateCompSupplyIndex(address(cTok
en));
1309                 for (uint j = 0; j < holders.lengt
h; j++) {
1310                     distributeSupplierComp(address
(cToken), holders[j], true);
1311                 }
1312             }
1313         }
1314     }
1315
1316     /** Comp Distribution Admin */
1317
1318     /**
1319      * @notice Set the amount of COMP distributed
per block

```



```

1296     * @param compRate_ The amount of COMP wei per
      block to distribute
1297     */
1298     function _setCompRate(uint compRate_) public {
1299         require(adminOrInitializing(), "only admin
      can change comp rate");
1300
1301         uint oldRate = compRate_;
1302         compRate = compRate_;
1303         emit NewCompRate(oldRate, compRate_);
1304
1305         refreshCompSpeedsInternal();
1306     }
1307
1308     /**
1309     * @notice Add markets to compMarkets, allowin
      g them to earn COMP in the flywheel
1310     * @param cTokens The addresses of the markets
      to add
1311     */
1312     function _addCompMarkets(address[] memory cTok
      ens) public {
1313         require(adminOrInitializing(), "only admin
      can add comp market");
1314
1315         for (uint i = 0; i < cTokens.length; i++)
1316         {
1317             _addCompMarketInternal(cTokens[i]);
1318         }
1319         refreshCompSpeedsInternal();
1320     }
1321
1322     function _addCompMarketInternal(address cToke
      n) internal {
1323         Market storage market = markets[cToken];
1324         require(market.isListed == true, "comp mar
      ket is not listed");
1325         require(market.isComped == false, "comp ma
      rket already added");
1326
1327         market.isComped = true;
1328         emit MarketComped(CToken(cToken), true);
1329
1330         if (compSupplyState[cToken].index == 0 &&
      compSupplyState[cToken].block == 0) {
1331             compSupplyState[cToken] = CompMarketSt
      ate({
1332                 index: compInitialIndex,
1333                 block: safe32(getBlockNumber(), "b
      lock number exceeds 32 bits")
1334             });
1335         }
1336
1337         if (compBorrowState[cToken].index == 0 &&
      compBorrowState[cToken].block == 0) {
1338             compBorrowState[cToken] = CompMarketSt
      ate({
1339                 index: compInitialIndex,
1340                 block: safe32(getBlockNumber(), "b
      lock number exceeds 32 bits")
1341             });
1342         }
1343     }
1344
1345     /**

```

```

1320     * @param compRate_ The amount of COMP wei per
      block to distribute
1321     */
1322     function _setCompRate(uint compRate_) public {
1323         require(adminOrInitializing(), "only admin
      can change comp rate");
1324
1325         uint oldRate = compRate_;
1326         compRate = compRate_;
1327         emit NewCompRate(oldRate, compRate_);
1328
1329         refreshCompSpeedsInternal();
1330     }
1331
1332     /**
1333     * @notice Add markets to compMarkets, allowin
      g them to earn COMP in the flywheel
1334     * @param cTokens The addresses of the markets
      to add
1335     */
1336     function _addCompMarkets(address[] memory cTok
      ens) public {
1337         require(adminOrInitializing(), "only admin
      can add comp market");
1338
1339         for (uint i = 0; i < cTokens.length; i++)
1340         {
1341             _addCompMarketInternal(cTokens[i]);
1342         }
1343         refreshCompSpeedsInternal();
1344     }
1345
1346     function _addCompMarketInternal(address cToke
      n) internal {
1347         Market storage market = markets[cToken];
1348         require(market.isListed == true, "comp mar
      ket is not listed");
1349         require(market.isComped == false, "comp ma
      rket already added");
1350
1351         market.isComped = true;
1352         emit MarketComped(CToken(cToken), true);
1353
1354         if (compSupplyState[cToken].index == 0 &&
      compSupplyState[cToken].block == 0) {
1355             compSupplyState[cToken] = CompMarketSt
      ate({
1356                 index: compInitialIndex,
1357                 block: safe32(getBlockNumber(), "b
      lock number exceeds 32 bits")
1358             });
1359         }
1360
1361         if (compBorrowState[cToken].index == 0 &&
      compBorrowState[cToken].block == 0) {
1362             compBorrowState[cToken] = CompMarketSt
      ate({
1363                 index: compInitialIndex,
1364                 block: safe32(getBlockNumber(), "b
      lock number exceeds 32 bits")
1365             });
1366         }
1367     }
1368
1369     /**

```

```

1346     * @notice Remove a market from compMarkets, p
reverting it from earning COMP in the flywheel
1347     * @param cToken The address of the market to
drop
1348     */
1349     function _dropCompMarket(address cToken) publi
c {
1350         require(msg.sender == admin, "only admin c
an drop comp market");
1351
1352         Market storage market = markets[cToken];
1353         require(market.isComped == true, "market i
s not a comp market");
1354
1355         market.isComped = false;
1356         emit MarketComped(cToken(cToken), false);
1357
1358         refreshCompSpeedsInternal();
1359     }
1360
1361     /**
1362     * @notice Return all of the markets
1363     * @dev The automatic getter may be used to ac
cess an individual market.
1364     * @return The list of market addresses
1365     */
1366     function getAllMarkets() public view returns
(CToken[] memory) {
1367         return allMarkets;
1368     }
1369
1370     function getBlockNumber() public view returns
(uint) {
1371         return block.number;
1372     }
1373
1374     /**
1375     * @notice Return the address of the COMP toke
n
1376     * @return The address of COMP
1377     */
1378     function getCompAddress() public view returns
(address) {
1379         return 0xc00e94Cb662C3520282E6f5717214004A
7f26888;
1380     }
1381 }
1382

```

```

1370     * @notice Remove a market from compMarkets, p
reverting it from earning COMP in the flywheel
1371     * @param cToken The address of the market to
drop
1372     */
1373     function _dropCompMarket(address cToken) publi
c {
1374         require(msg.sender == admin, "only admin c
an drop comp market");
1375
1376         Market storage market = markets[cToken];
1377         require(market.isComped == true, "market i
s not a comp market");
1378
1379         market.isComped = false;
1380         emit MarketComped(cToken(cToken), false);
1381
1382         refreshCompSpeedsInternal();
1383     }
1384
1385     /**
1386     * @notice Return all of the markets
1387     * @dev The automatic getter may be used to ac
cess an individual market.
1388     * @return The list of market addresses
1389     */
1390     function getAllMarkets() public view returns
(CToken[] memory) {
1391         return allMarkets;
1392     }
1393
1394     function getBlockNumber() public view returns
(uint) {
1395         return block.number;
1396     }
1397
1398     /**
1399     * @notice Return the address of the COMP toke
n
1400     * @return The address of COMP
1401     */
1402     function getCompAddress() public view returns
(address) {
1403         return 0x0Ed0Ca6872073E02cd3aE005BaF04bA43
BE947fA;
1404     }
1405 }
1406

```