KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	20210226	V1.0

Report Information

Title	Version	Document Number	Type
Horizon Protocol Smart Contract Audit Report	V1.0		Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from From February 22, 2021 to February 26, 2021 . During this period, the security and standardization of **the smart contract code of the Horizon Protocol** will be audited and used as the statistical basis for the report.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). **The smart contract code of the Horizon Protocol** is comprehensively assessed as **SAFE**.

Results of this smart contract security audit： SAFE

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

Report information of this audit:

Report Number：

Report query address link:

https://attest.im/attestation/searchResult?qurey=

Target information of the Horizon Protocol audit:

Target information	
Token name	HZN
Token address	https://github.com/Horizon-Protocol/Horizon-Smart-Contract https://github.com/Horizon-Protocol/horizon-utility
Code type	Token code, Binance Smart Chain smart contract code
Code language	solidity

Contract documents and hash:

Contract documents	MD5
AddressResolver.sol	04D1F8A94BFC8469C683A60DB742618C
AddressSetLib.sol	545EA44D9E71C4656E2844FCBD197D16
BaseSynthetix.sol	4F4964E8B2F03B7C47C0746191BEBBE5
BinaryOption.sol	765A09978FA894AAD8F5377F2101F3E2
BinaryOptionMarket.sol	6A7F7DB0E7304A7540A7FB33C6C5C879
BinaryOptionMarketData.sol	4CB3BCA24B00460A0BE6AD739757E8F7
BinaryOptionMarketFactory.sol	FF0446AE2CA948BBF437F94543DE4B72
BinaryOptionMarketManager.sol	0120DCCEB1D5B6FA2839C1889E1D7AFB
ContractStorage.sol	C8486044576D7E4A581F030C6FFFA6D1
DappMaintenance.sol	8F65E30EF119C34DBCDB79E65B3F7D23
DebtCache.sol	BF2784C3FAF05415F30FBD6427052693
DelegateApprovals.sol	F030744E7F0024D02226DB2EB4D7DDAC
Depot.sol	1D0D3A9A9FB73CBDA9E68A8973D9FA6D
EmptyEtherCollateral.sol	C833F7FAFC01C853A28BE895E8D987D0
EscrowChecker.sol	2327838B777BDB2549F0BD10A0E5A894
EternalStorage.sol	F182FD0A865FE33035727AF242441D65
EtherCollateral.sol	2A61584AE6E62FFC4ECBE26FB793E970

EtherCollateralsUSD.sol	407985977D0507A8946B19D6594D4875
Exchanger.sol	41588AA9F5898592E2C0F8C194B16B7B
ExchangeRates.sol	477E2BF1B3BE7AACC5F06C5511014AD4
ExchangerWithVirtualSynth.sol	45FBF57100390C1CF57B9D6A43920502
ExchangeState.sol	1FB8CEF573F63C0B76223B1DF535EB76
ExternStateToken.sol	6D17B77ACBB2D96683DF731B3BC8FFE1
FeePool.sol	BC3D95852FDC523CE1DE7E41B514C74C
FeePoolEternalStorage.sol	4FD739012DFE62E9C1F3C9FF36275084
FeePoolState.sol	FDD861CFDA41C23663DA5292981FF582
FixedSupplySchedule.sol	ED5908F05B81E2855C251DC0A9F5FA05
FlexibleStorage.sol	21ED6B5F06955EA35303B9FE11F73321
IAddressResolver.sol	BBFB2F571C6DCD219D78BE0CCDEA0AD6
IBinaryOption.sol	03785F53AB5B8FE215D6153F3E44BBBE
IBinaryOptionMarket.sol	102E725379497DDFAFC5B005B344040B
IBinaryOptionMarketManager.sol	CD87B96D47649430272AC4A840301C4B
IDebtCache.sol	7384AED10533815C5E904196B9F24CDF
IDelegateApprovals.sol	B86ED945D96DBB47AB7E7523D341DF67
IDepot.sol	FC7F67468ED277B3599137F304D29391

IERC20.sol	97FC6B90051EAABEC6C182164D02E1AC
IEtherCollateral.sol	4A59CBDF416EA08927E73CEDE3909CA0
IEtherCollateralsUSD.sol	7C49B5CF85A52CD5FCA499E9BCA81080
IExchanger.sol	4FF110F5FBFE1EA60D2F224FC33A21EA
IExchangeRates.sol	1857CCBFC3BC2E51C80364C378615440
IExchangeState.sol	21401467515FCE2CD9F684A780D3B44F
IFeePool.sol	5F5811A1D88EED54DA5A8EDEA535301F
IFlexibleStorage.sol	B5EC53D32F1EE9360D1F83381C96FFDA
IHasBalance.sol	341F3C1B7AFEDAE29AA5E8F536C3060E
IIssuer.sol	3CFAF98451E3EABE7D8B9A3B3B7F9DB2
ILiquidations.sol	F6FE020FEBF5CE168DED18B801280D0B
IRewardEscrow.sol	BEFAE97F54CC924E5027928F1F9862B6
IRewardsDistribution.sol	D4AD6038AE8D5AC5E0BA623AE220A0B7
IStakingRewards.sol	62F7B328BBF956E7AF5E2DE45890B3EB
ISupplySchedule.sol	424A6C206D72C5F7DC2542C441765AA7
ISynth.sol	E901D166D20879951041A34EB1D55054
ISynthetix.sol	4B1B8519D2DCEE059C331C8097BD388D
ISynthetixBridgeToBase.sol	47357406B973C5F0EF519BA2FAAFE6EB
ISynthetixBridgeToOptimism.sol	226F53815B37BE4C9702CF31E3EFA762

ISynthetixState.sol	18B96DC4DCD420BD4D85F4EF1E40738D
ISystemSettings.sol	EF92B89997298D94D4A89DB8461CA2EC
ISystemStatus.sol	A6FD949EF03512F5845421630BD251B0
ITradingRewards.sol	86864EC1BA88F0044389AF3C93EB912C
IVirtualSynth.sol	7D1CE45C90D82C68FDA7E2F35C0A4B05
Issuer.sol	4165E3E68A4B99A1BF32F98D1B9B4B60
LimitedSetup.sol	F04C3705618BC28B8B2A379F7DA8B704
Liquidations.sol	EBA6D808EF1DD285362E69D4C331325B
Math.sol	93199516B56648397A09441E0EB67658
MintableSynthetix.sol	E4EDE1F68808AB806A870787B16EF544
MixinResolver.sol	FCBA13CA26EC7C0BC8D27F667B6394E8
MixinSystemSettings.sol	AA4786C3B0AE8F5C8C1A06050DACF4EA
MultiCollateralSynth.sol	021AC479D79E101DDE38E931CDFB1E66
Owned.sol	C715A1EACF80C82FAB7CE272C4FBCEBD
Pausable.sol	C7C3F173119B1F70FB454C14C5197A04
Proxy.sol	96FCD783BAF09458D0BA2872FA3EE60D
Proxyable.sol	D15E286292AB37A49D276309C6F47B66
ProxyERC20.sol	6FD8FE5C6040DAD710A4D448BF6D26DB
PurgeableSynth.sol	A721A6232FDCC51BBF95103651B57742
ReadProxy.sol	F610D1CE029407CFF7D496E9021E7E4A

RealtimeDebtCache.sol	253065BDFDCD7FE6293672A01593E960
RewardEscrow.sol	50D81EB1AB2D5BDD95A842CB12612E3F
RewardsDistribution.sol	CCB785BFAB190D63F402DBB3527B903B
RewardsDistributionRecipient.sol	9F3579B5DA821EE5B3EA288C8A8D6F96
SafeDecimalMath.sol	98DD41B8BCE1871536E8A31D0D44FFC2
StakingRewards.sol	CF6D1BBF9DD36B09E925F26E1B63DC6E
State.sol	3E6979FB59864A0FA4C73DCEBB70EFE0
SupplySchedule.sol	F3722BE8712F32F4A2A140190ABC7A42
Synth.sol	08B009DA78CA749A057EFD0F8E6E90D0
Synthetix.sol	C56881C477EC3B221C6ED4AB3DF1E35B
SynthetixBridgeToBase.sol	DBE192A04044938F6586FC5C55AB2FF1
SynthetixBridgeToOptimism.sol	6594B6C514BFE30689A675506A58C2F6
SynthetixEscrow.sol	9EF5E937FCECC665831D78374E8CA7DF
SynthetixState.sol	1B8305C81852137F61680BAE069F7D7E
SynthUtil.sol	D2FA4D882DCDDEAA60A912A53FCE1B01
SystemSettings.sol	663B8E4D2F32B66A4CEDD0F7120FBCC3
SystemStatus.sol	6E25403C48C330D6D4487C2DCF4F3979
FakeTradingRewards.sol	D5AA5F21F91235696FF363F44921773D
GenericMock.sol	2E6E4286E99D73B4493A239DD7554794

MockAggregatorV2V3.sol	66E03FF30EC8BA604C9BB70389C920AD
MockBinaryOptionMarket.sol	7965268D14BA03475847B984EAAB3BD1
MockBinaryOptionMarketManager.sol	6239246B177ED45B40B9FC6A1C6CBA4A
MockContractStorage.sol	95D154DE2AF3DD2078F7ABC3DCCF0926
MockEtherCollateral.sol	F5C000363E51B05A96FD7D7288B82F42
MockExchanger.sol	7A44B54A271B62F6F1D83A4B9B0479FC
MockFlagsInterface.sol	E5AFF7FBE8153986477B12200E6CEBA2
MockMintableSynthetix.sol	27D5AB4D3EEB3073F4C366680A8A14D1
MockMutator.sol	DCBF64B5DA84DE33070A6DB4615658CB
MockRewardsRecipient.sol	B37021E017E97D07573C965D3E2B6B89
MockSynth.sol	D30DD99CC3901BC4EB262D96F0409F19
OneWeekSetup.sol	992FC2DAA72E149CE8193BE2D86AC12F
PublicEST.sol	588E3E42C0A570D62EF0691944172C7E
PublicMath.sol	936CADBA3F5FA008FBF27882CFDC7D05
PublicSafeDecimalMath.sol	C9DCFE9642FC913D0563900C00494014
SwapWithVirtualSynth.sol	660904AB3FD4013CD315104BE9F27D70
TestableAddressSet.sol	1962F24227DBB1BFD697A83EF361E33E

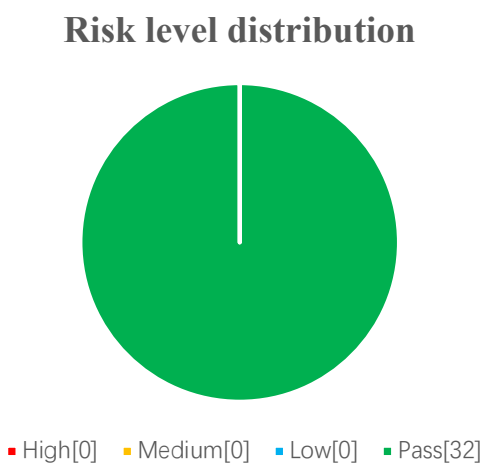
TestableBinaryOptionMarket.sol	D6B538357B747D562B0D69BFAA546F83
TestableDebtCache.sol	AF7BDFF52F79CC7448BF943C8C18F5CF
TestableMixinResolver.sol	AAF7D06C46271BC21C55C0D20E4F0E3E
TestablePausable.sol	6BEFF6A5EB7A4E460DC14679E395796F
TestableState.sol	01ED73336C57F61889A4C9E4878AAB76
TokenExchanger.sol	B863B6D99C4F318B74215A05458011C1
UsingReadProxy.sol	D3648527B17EA16752A0D73F5779F112
TokenState.sol	E4FBB5935F8DA609E16739FA4DC8CE46
TradingRewards.sol	6C2C7EBA1B9FE2020A28DDEBF3B740E9
VirtualSynth.sol	2D67A4953C2776ADA509C065FA079D16
BinaryOptionMarketData.sol	C7A87E7D6810EBE80D69976BE1706311
SynthSummaryUtil.sol	04D1F8A94BFC8469C683A60DB742618C

2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level：

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	0	32



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**HZN token standard contract**	Pass	After testing, there is no such safety vulnerability.
	Exchange rate management contract data interface	Pass	After testing, there is no such safety vulnerability.
	Oracle address setting function	Pass	After testing, there is no such safety vulnerability.
	Price feed function	Pass	After testing, there is no such safety vulnerability.
	User entity queue control	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	**Compiler version security**	Pass	After testing, there is no such safety vulnerability.
	Redundant code	Pass	After testing, there is no such safety vulnerability.
	Use of safe arithmetic library	Pass	After testing, there is no such safety vulnerability.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.
	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.

	tx.oriigin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.
	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.
	call injection attack	Pass	After testing, there is no such safety vulnerability.
	Low-level function safety	Pass	After testing, there is no such safety vulnerability.
	Vulnerability of additional token issuance	Pass	After testing, there is no such safety vulnerability.
	Access control defect detection	Pass	After testing, there is no such safety vulnerability.
	Numerical overflow detection	Pass	After testing, there is no such safety vulnerability.
	Arithmetic accuracy error	Pass	After testing, there is no such safety vulnerability.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Unsafe interface use	Pass	After testing, there is no such safety vulnerability.
	Variable coverage	Pass	After testing, there is no such safety vulnerability.
	Uninitialized storage pointer	Pass	After testing, there is no such safety vulnerability.

	Return value call verification	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Pass	After testing, there is no such safety vulnerability.
	Timestamp dependent attack	Pass	After testing, there is no such safety vulnerability.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	Reentry attack detection	Pass	After testing, there is no such safety vulnerability.
	Replay attack detection	Pass	After testing, there is no such safety vulnerability.
	Rearrangement attack detection	Pass	After testing, there is no such safety vulnerability.

3. Analysis of code audit results

3.1. HZN token standard contract【PASS】

Audit analysis: The project contract uses BaseSynthetix.sol as the token standard contract, and the synthetic asset is completed on the basis of the HZN token. Users generate synthetic assets by staking HZN tokens, which replicate the price of real currency, operate based on the Binance Smart Chain, and have all the standards of BEP-20 tokens. After audit, the contract function design is reasonable and the authority control is correct.

```
contract BaseSynthetix is IERC20, ExternStateToken, MixinResolver, ISynthetix {
// ========== STATE VARIABLES ==========

// Available Synths which can be used with the system
string public constant TOKEN_NAME = "Phoenix Horizon";
string public constant TOKEN_SYMBOL = "HZN";// knownsec  token symbol
uint8 public constant DECIMALS = 18;
bytes32 public constant zUSD = "zUSD";

/* ========== ADDRESS RESOLVER CONFIGURATION ========== */

bytes32 private constant CONTRACT_SYNTHETIXSTATE = "SynthetixState";
bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
bytes32 private constant CONTRACT_ISSUER = "Issuer";
bytes32 private constant CONTRACT_SUPPLYSCHEDULE = "SupplySchedule";
bytes32 private constant CONTRACT_REWARDSDISTRIBUTION = "RewardsDistribution";

bytes32[24] private addressesToCache = [
    CONTRACT_SYSTEMSTATUS,
    CONTRACT_EXCHANGER,
    CONTRACT_ISSUER,
```

```
            CONTRACT_SUPPLYSCHEDULE,
            CONTRACT_REWARDSDISTRIBUTION,
            CONTRACT_SYNTHETIXSTATE
        ];

        // ========== CONSTRUCTOR ==========

        constructor(
            address payable _proxy,
            TokenState _tokenState,
            address _owner,
            uint _totalSupply,
            address _resolver
        )
            public
            ExternStateToken(_proxy, _tokenState, TOKEN_NAME, TOKEN_SYMBOL,
_totalSupply, DECIMALS, _owner)
            MixinResolver(_resolver, addressesToCache)
        {}
    ...
    ...
```

Recommendation： nothing.

3.2. Exchange rate management contract data interface 【PASS】

Audit analysis: The IStdReference interface is newly added to the project contract as a standard structure for data return, which contains three parts of data: rate, lastUpdatedBase, and lastUpdatedQuote. After audit, the interface data structure is reasonably designed and the access control is correct.

```
interface IStdReference {// knownsec Reference structure return interface
    // A structure returned whenever someone requests for standard reference data.
    struct ReferenceData {// knownsec ReferenceData structure
        uint256 rate; // base/quote exchange rate, multiplied by 1e18.
        uint256 lastUpdatedBase; // UNIX epoch of the last time when base price gets updated.
        uint256 lastUpdatedQuote; // UNIX epoch of the last time when quote price gets updated.
    }

    // Returns the price data for the given base/quote pair. Revert if not available.
    function getReferenceData(string calldata _base, string calldata _quote) external view returns (ReferenceData memory);

    // Similar to getReferenceData, but with multiple base/quote pairs at once.
    function getReferenceDataBulk(string[] calldata _bases, string[] calldata _quotes)
        external
        view
        returns (ReferenceData[] memory);// knownsec Use Reference array to return externally
}
```

Recommendation： nothing.

3.3. Oracle address setting function【PASS】

Audit analysis: The bandProtocolOracle variable is added to the project contract as the oracle address of the band. At the same time, the Owner can use the setBandProtocolOracle method to set the oracle address. After auditing, the interface data structure is reasonable and the access control is correct.

```
function setBandProtocolOracle(IStdReference _bandProtocolOracle) external onlyOwner {// knownsec Band oracle address setting owner is available
        bandProtocolOracle = _bandProtocolOracle;
        emit BandProtocolOracleUpdated(bandProtocolOracle);// knownsec Event record
    }
```

Recommendation： nothing.

3.4. Price feed function【PASS】

Audit analysis: The project contract uses _getRateAndUpdatedTime to update the price, and uses getReferenceData to obtain the predicted price. After audit, the interface data structure is reasonably designed and the access control is correct.

```
function _getRateAndUpdatedTime(bytes32 currencyKey) internal view returns (RateAndUpdatedTime memory) {// knownsec Internal call to construct rate and time structure
        // AggregatorV2V3Interface aggregator = aggregators[currencyKey];
        ...
        ...
        // TODO change HZN Token's price feed for testnet
        if (bandProtocolOracle != IStdReference(0) && currencyKey != HZN) {// knownsec Oracle address and logo check
            // remove asset prefix
            uint8 offset = 1;
            // pass remove prefix for HZN currencyKey
            if (currencyKey == HZN) {
                currencyKey = SNX;
                offset = 0;
```

```
                }
                string memory stringCurrencyKey = bytes32ToString(currencyKey, offset);// knownsec Calculation
                IStdReference.ReferenceData memory answer = bandProtocolOracle.getReferenceData(stringCurrencyKey, "USD");// knownsec Oracle information exchange
                uint256 updatedAt = answer.lastUpdatedBase >= answer.lastUpdatedQuote
                    ? answer.lastUpdatedBase
                    : answer.lastUpdatedQuote;
                uint roundId = currentRoundForRate[currencyKey];// knownsec Round acquisition
                return
                    RateAndUpdatedTime({
                        rate: uint216(_rateOrInverted(currencyKey, _formatAggregatorAnswer(currencyKey, answer.rate), roundId)),
                        time: uint40(updatedAt)
                    });// knownsec Return rate and time structure
            } else {
                uint roundId = currentRoundForRate[currencyKey];// knownsec Round acquisition
                RateAndUpdatedTime memory entry = _rates[currencyKey][roundId];

                return RateAndUpdatedTime({rate: uint216(_rateOrInverted(currencyKey, entry.rate, roundId)), time: entry.time});// knownsec Return rate and time structure
            }
        }
```

Recommendation： nothing.

3.5. User entity queue control【PASS】

Audit analysis: In the project contract, the maximum user entity is modified to 0, which will cause AssociatedContract to be unable to use the appendExchangeEntry

method to add user entities, but since the owner can use setMaxEntriesInQueue to adjust this variable, the rating is passed.

```
contract ExchangeState is Owned, State, IExchangeState {
    mapping(address => mapping(bytes32 => IExchangeState.ExchangeEntry[])) public exchanges;

    uint public maxEntriesInQueue = 0;// knownsec Initialization does not allow user entities, and subsequent owners can be modified with setMaxEntriesInQueue

    constructor(address _owner, address _associatedContract) public Owned(_owner) State(_associatedContract) {}

    /* ========== SETTERS ========== */

    function setMaxEntriesInQueue(uint _maxEntriesInQueue) external onlyOwner {
        maxEntriesInQueue = _maxEntriesInQueue;
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function appendExchangeEntry(
        address account,
        bytes32 src,
        uint amount,
        bytes32 dest,
        uint amountReceived,
        uint exchangeFeeRate,
        uint timestamp,
        uint roundIdForSrc,
        uint roundIdForDest
    ) external onlyAssociatedContract {
        require(exchanges[account][dest].length < maxEntriesInQueue, "Max queue length reached");
```

```
        exchanges[account][dest].push(
            ExchangeEntry({
                src: src,
                amount: amount,
                dest: dest,
                amountReceived: amountReceived,
                exchangeFeeRate: exchangeFeeRate,
                timestamp: timestamp,
                roundIdForSrc: roundIdForSrc,
                roundIdForDest: roundIdForDest
            })
        );
    }
```

Recommendation： nothing.

4. Basic code vulnerability detection

4.1. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

Audit result: After testing, the smart contract code has formulated the compiler version 0.6.0 within the major version, and there is no such security problem.

Recommendation： nothing.

4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.3. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

Audit result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Recommendation： nothing.

4.4. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.5. Reasonable use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.6. Fallback function safety【PASS】

Check whether the fallback function is used correctly in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.7. tx.origin authentication【PASS】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.8. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.9. Gas consumption detection【PASS】

Check whether the consumption of gas exceeds the maximum block limit.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.10. call injection attack【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

Audit result: After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation： nothing.

4.11. Low-level function safety【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.12. Vulnerability of additional token issuance【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

Audit result: After testing, the smart contract code has the function of issuing additional tokens, but it is only used for the constructor, so it is passed.

Recommendation： nothing.

4.13. Access control defect detection【PASS】

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.14. Numerical overflow detection【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.15. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary

programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.17. Unsafe interface usage【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.18. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Audit result: After testing, the smart contract code does not use structure, there is no such problem.

Recommendation： nothing.

4.20. Return value call verification【PASS】

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send BNB to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling (can be Limit by passing in gas_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to BNB sending failure.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.21. Transaction order dependency【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Audit result: After testing, the _approve function in the contract has a transaction sequence dependency attack risk, but the vulnerability is extremely difficult to exploit, so it is rated as passed. The code is as follows:

```
function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");
        _allowances[owner][spender] = amount;//knownnsec// Transaction order depends on
risk        emit Approval(owner, spender, amount);
    }
```

The possible security risks are described as follows:

1. By calling the approve function, user A allows user B to transfer money on his behalf to N (N>0);

2. After a period of time, user A decides to change N to M (M>0), so call the approve function again;

3. User B quickly calls the transferFrom function to transfer N number of tokens before the second call is processed by the miner;

4. After user A's second call to approve is successful, user B can obtain M's transfer quota again, that is, user B obtains N+M's transfer quota through the transaction sequence attack.

Recommendation：

1. Front-end restriction, when user A changes the quota from N to M, he can first change from N to 0, and then from 0 to M.

2. Add the following code at the beginning of the approve function:

require((_value == 0) || (allowed[msg.sender][_spender] == 0));

4.22. Timestamp dependency attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.23. Denial of service attack【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.24. Fake recharge vulnerability【PASS】

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.25. Reentry attack detection【PASS】

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send BNB. When the **call.value()** function to send BNB occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

Audit results： After auditing, the vulnerability does not exist in the smart contract code.

Recommendation： nothing.

4.26. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

Audit results： After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation： nothing.

4.27. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping,

so that the attacker has the opportunity to store their own information in the contract. in.

Audit results： After auditing, the vulnerability does not exist in the smart contract code.

Recommendation： nothing.

5. Appendix A：Contract code

Source code：

AddressResolver.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";

import "./interfaces/IAddressResolver.sol";
import "./interfaces/IIssuer.sol";


// https://docs.synthetix.io/contracts/source/contracts/addressresolver
contract AddressResolver is Owned, IAddressResolver {
    mapping(bytes32 => address) public repository;

    constructor(address _owner) public Owned(_owner) {}

    /* ========== MUTATIVE FUNCTIONS ========== */

    function importAddresses(bytes32[] calldata names, address[] calldata destinations) external onlyOwner {
        require(names.length == destinations.length, "Input lengths must match");

        for (uint i = 0; i < names.length; i++) {
            repository[names[i]] = destinations[i];
        }
    }

    /* ========== VIEWS ========== */

    function getAddress(bytes32 name) external view returns (address) {
        return repository[name];
    }

    function requireAndGetAddress(bytes32 name, string calldata reason) external view returns (address) {
        address _foundAddress = repository[name];
        require(_foundAddress != address(0), reason);
        return _foundAddress;
    }

    function getSynth(bytes32 key) external view returns (address) {
        IIssuer issuer = IIssuer(repository["Issuer"]);
        require(address(issuer) != address(0), "Cannot find Issuer address");
        return address(issuer.synths(key));
    }
}
```

AddressSetLib.sol

```
pragma solidity ^0.5.16;


// https://docs.synthetix.io/contracts/source/libraries/addresssetlib/
library AddressSetLib {
    struct AddressSet {
        address[] elements;
        mapping(address => uint) indices;
    }

    function contains(AddressSet storage set, address candidate) internal view returns (bool) {
        if (set.elements.length == 0) {
            return false;
        }
        uint index = set.indices[candidate];
        return index != 0 || set.elements[0] == candidate;
    }

    function getPage(
        AddressSet storage set,
        uint index,
        uint pageSize
    ) internal view returns (address[] memory) {
        // NOTE: This implementation should be converted to slice operators if the compiler is updated to v0.6.0+
        uint endIndex = index + pageSize; // The check below that endIndex <= index handles overflow.

        // If the page extends past the end of the list, truncate it.
        if (endIndex > set.elements.length) {
            endIndex = set.elements.length;
```

```
            }
            if (endIndex <= index) {
                return new address[](0);
            }

            uint n = endIndex - index; // We already checked for negative overflow.
            address[] memory page = new address[](n);
            for (uint i; i < n; i++) {
                page[i] = set.elements[i + index];
            }
            return page;
        }

        function add(AddressSet storage set, address element) internal {
            // Adding to a set is an idempotent operation.
            if (!contains(set, element)) {
                set.indices[element] = set.elements.length;
                set.elements.push(element);
            }
        }

        function remove(AddressSet storage set, address element) internal {
            require(contains(set, element), "Element not in set.");
            // Replace the removed element with the last element of the list.
            uint index = set.indices[element];
            uint lastIndex = set.elements.length - 1; // We required that element is in the list, so it is not empty.
            if (index != lastIndex) {
                // No need to shift the last element if it is the one we want to delete.
                address shiftedElement = set.elements[lastIndex];
                set.elements[index] = shiftedElement;
                set.indices[shiftedElement] = index;
            }
            set.elements.pop();
            delete set.indices[element];
        }
    }
```

BaseSynthetix.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./interfaces/IERC20.sol";
import "./ExternStateToken.sol";
import "./MixinResolver.sol";
import "./interfaces/ISynthetix.sol";

// Internal references
import "./interfaces/ISynth.sol";
import "./TokenState.sol";
import "./interfaces/ISynthetixState.sol";
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IExchanger.sol";
import "./interfaces/IIssuer.sol";
import "./SupplySchedule.sol";
import "./interfaces/IRewardsDistribution.sol";
import "./interfaces/IVirtualSynth.sol";


contract BaseSynthetix is IERC20, ExternStateToken, MixinResolver, ISynthetix {
    // ========== STATE VARIABLES ==========

    // Available Synths which can be used with the system
    string public constant TOKEN_NAME = "Horizon Protocol";
    string public constant TOKEN_SYMBOL = "HZN";
    uint8 public constant DECIMALS = 18;
    bytes32 public constant zUSD = "zUSD";

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYNTHETIXSTATE = "SynthetixState";
    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_SUPPLYSCHEDULE = "SupplySchedule";
    bytes32 private constant CONTRACT_REWARDSDISTRIBUTION = "RewardsDistribution";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_EXCHANGER,
        CONTRACT_ISSUER,
        CONTRACT_SUPPLYSCHEDULE,
        CONTRACT_REWARDSDISTRIBUTION,
        CONTRACT_SYNTHETIXSTATE
    ];
```

```
    // ========== CONSTRUCTOR ==========

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        address _owner,
        uint _totalSupply,
        address _resolver
    )
        public
        ExternStateToken(_proxy, _tokenState, TOKEN_NAME, TOKEN_SYMBOL, _totalSupply, DECIMALS,
_owner)
        MixinResolver(_resolver, addressesToCache)
    {}

    /* ========== VIEWS ========== */

    function synthetixState() internal view returns (ISynthetixState) {
        return ISynthetixState(requireAndGetAddress(CONTRACT_SYNTHETIXSTATE, "Missing HorizonState
address"));
    }

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus
address"));
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function supplySchedule() internal view returns (SupplySchedule) {
        return SupplySchedule(requireAndGetAddress(CONTRACT_SUPPLYSCHEDULE, "Missing
SupplySchedule address"));
    }

    function rewardsDistribution() internal view returns (IRewardsDistribution) {
        return
            IRewardsDistribution(requireAndGetAddress(CONTRACT_REWARDSDISTRIBUTION, "Missing
RewardsDistribution address"));
    }

    function debtBalanceOf(address account, bytes32 currencyKey) external view returns (uint) {
        return issuer().debtBalanceOf(account, currencyKey);
    }

    function totalIssuedSynths(bytes32 currencyKey) external view returns (uint) {
        return issuer().totalIssuedSynths(currencyKey, false);
    }

    function totalIssuedSynthsExcludeEtherCollateral(bytes32 currencyKey) external view returns (uint) {
        return issuer().totalIssuedSynths(currencyKey, true);
    }

    function availableCurrencyKeys() external view returns (bytes32[] memory) {
        return issuer().availableCurrencyKeys();
    }

    function availableSynthCount() external view returns (uint) {
        return issuer().availableSynthCount();
    }

    function availableSynths(uint index) external view returns (ISynth) {
        return issuer().availableSynths(index);
    }

    function synths(bytes32 currencyKey) external view returns (ISynth) {
        return issuer().synths(currencyKey);
    }

    function synthsByAddress(address synthAddress) external view returns (bytes32) {
        return issuer().synthsByAddress(synthAddress);
    }

    function isWaitingPeriod(bytes32 currencyKey) external view returns (bool) {
        return exchanger().maxSecsLeftInWaitingPeriod(messageSender, currencyKey) > 0;
    }

    function anySynthOrSNXRateIsInvalid() external view returns (bool anyRateInvalid) {
        return issuer().anySynthOrSNXRateIsInvalid();
    }

    function maxIssuableSynths(address account) external view returns (uint maxIssuable) {
```

```
        return issuer().maxIssuableSynths(account);
    }

    function remainingIssuableSynths(address account)
        external
        view
        returns (
            uint maxIssuable,
            uint alreadyIssued,
            uint totalSystemDebt
        )
    {
        return issuer().remainingIssuableSynths(account);
    }

    function collateralisationRatio(address _issuer) external view returns (uint) {
        return issuer().collateralisationRatio(_issuer);
    }

    function collateral(address account) external view returns (uint) {
        return issuer().collateral(account);
    }

    function transferableSynthetix(address account) external view returns (uint transferable) {
        (transferable, ) = issuer().transferableSynthetixAndAnyRateIsInvalid(account,
tokenState.balanceOf(account));
    }

    function _canTransfer(address account, uint value) internal view returns (bool) {
        (uint initialDebtOwnership, ) = synthetixState().issuanceData(account);

        if (initialDebtOwnership > 0) {
            (uint transferable, bool anyRateIsInvalid) = issuer().transferableSynthetixAndAnyRateIsInvalid(
                account,
                tokenState.balanceOf(account)
            );
            require(value <= transferable, "Cannot transfer staked or escrowed HZN");
            require(!anyRateIsInvalid, "A zasset or HZN rate is invalid");
        }
        return true;
    }

    // ========== MUTATIVE FUNCTIONS ==========

    function transfer(address to, uint value) external optionalProxy systemActive returns (bool) {
        // Ensure they're not trying to exceed their locked amount -- only if they have debt.
        _canTransfer(messageSender, value);

        // Perform the transfer: if there is a problem an exception will be thrown in this call.
        _transferByProxy(messageSender, to, value);

        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint value
    ) external optionalProxy systemActive returns (bool) {
        // Ensure they're not trying to exceed their locked amount -- only if they have debt.
        _canTransfer(from, value);

        // Perform the transfer: if there is a problem,
        // an exception will be thrown in this call.
        return _transferFromByProxy(messageSender, from, to, value);
    }

    function issueSynths(uint amount) external issuanceActive optionalProxy {
        return issuer().issueSynths(messageSender, amount);
    }

    function issueSynthsOnBehalf(address issueForAddress, uint amount) external issuanceActive optionalProxy {
        return issuer().issueSynthsOnBehalf(issueForAddress, messageSender, amount);
    }

    function issueMaxSynths() external issuanceActive optionalProxy {
        return issuer().issueMaxSynths(messageSender);
    }

    function issueMaxSynthsOnBehalf(address issueForAddress) external issuanceActive optionalProxy {
        return issuer().issueMaxSynthsOnBehalf(issueForAddress, messageSender);
    }

    function burnSynths(uint amount) external issuanceActive optionalProxy {
        return issuer().burnSynths(messageSender, amount);
    }
```

```
function burnSynthsOnBehalf(address burnForAddress, uint amount) external issuanceActive optionalProxy {
        return issuer().burnSynthsOnBehalf(burnForAddress, messageSender, amount);
}

function burnSynthsToTarget() external issuanceActive optionalProxy {
        return issuer().burnSynthsToTarget(messageSender);
}

function burnSynthsToTargetOnBehalf(address burnForAddress) external issuanceActive optionalProxy {
        return issuer().burnSynthsToTargetOnBehalf(burnForAddress, messageSender);
}

function exchange(
        bytes32,
        uint,
        bytes32
) external returns (uint) {
        _notImplemented();
}

function exchangeOnBehalf(
        address,
        bytes32,
        uint,
        bytes32
) external returns (uint) {
        _notImplemented();
}

function exchangeWithTracking(
        bytes32,
        uint,
        bytes32,
        address,
        bytes32
) external returns (uint) {
        _notImplemented();
}

function exchangeOnBehalfWithTracking(
        address,
        bytes32,
        uint,
        bytes32,
        address,
        bytes32
) external returns (uint) {
        _notImplemented();
}

function exchangeWithVirtual(
        bytes32,
        uint,
        bytes32,
        bytes32
) external returns (uint, IVirtualSynth) {
        _notImplemented();
}

function settle(bytes32)
        external
        returns (
                uint,
                uint,
                uint
        )
{
        _notImplemented();
}

function mint() external returns (bool) {
        _notImplemented();
}

function liquidateDelinquentAccount(address, uint) external returns (bool) {
        _notImplemented();
}

function mintSecondary(address, uint) external {
        _notImplemented();
}

function mintSecondaryRewards(uint) external {
        _notImplemented();
}

function burnSecondary(address, uint) external {
```

```
        _notImplemented();
    }

    function _notImplemented() internal pure {
        revert("Cannot be run on this layer");
    }

    // ========== MODIFIERS ==========

    modifier onlyExchanger() {
        require(msg.sender == address(exchanger()), "Only Exchanger can invoke this");
        _;
    }

    modifier systemActive() {
        systemStatus().requireSystemActive();
        _;
    }

    modifier issuanceActive() {
        systemStatus().requireIssuanceActive();
        _;
    }

    modifier exchangeActive(bytes32 src, bytes32 dest) {
        systemStatus().requireExchangeActive();
        systemStatus().requireSynthsActive(src, dest);
        _;
    }
}
```

BinaryOption.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./interfaces/IERC20.sol";
import "./interfaces/IBinaryOption.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./BinaryOptionMarket.sol";


// https://docs.synthetix.io/contracts/source/contracts/binaryoption
contract BinaryOption is IERC20, IBinaryOption {
    /* ========== LIBRARIES ========== */

    using SafeMath for uint;
    using SafeDecimalMath for uint;

    /* ========== STATE VARIABLES ========== */

    string public constant name = "HZN Binary Option";
    string public constant symbol = "hOPT";
    uint8 public constant decimals = 18;

    BinaryOptionMarket public market;

    mapping(address => uint) public bidOf;
    uint public totalBids;

    mapping(address => uint) public balanceOf;
    uint public totalSupply;

    // The argument order is allowance[owner][spender]
    mapping(address => mapping(address => uint)) public allowance;

    // Enforce a 1 cent minimum bid balance
    uint internal constant _MINIMUM_BID = 1e16;

    /* ========== CONSTRUCTOR ========== */

    constructor(address initialBidder, uint initialBid) public {
        market = BinaryOptionMarket(msg.sender);
        bidOf[initialBidder] = initialBid;
        totalBids = initialBid;
    }

    /* ========== VIEWS ========== */

    function _claimableBalanceOf(
        uint _bid,
```

```
        uint price,
        uint exercisableDeposits
    ) internal view returns (uint) {
        uint owed = _bid.divideDecimal(price);
        uint supply = _totalClaimableSupply(exercisableDeposits);

        /* The last claimant might be owed slightly more or less than the actual remaining deposit
           based on rounding errors with the price.
           Therefore if the user's bid is the entire rest of the pot, just give them everything that's left.
           If there is no supply, then this option lost, and we'll return 0.
           */
        if ((_bid == totalBids && _bid != 0) || supply == 0) {
            return supply;
        }

        /* Note that option supply on the losing side and deposits can become decoupled,
           but losing options are not claimable, therefore we only need to worry about
           the situation where supply < owed on the winning side.

           If somehow a user who is not the last bidder is owed more than what's available,
           subsequent bidders will be disadvantaged. Given that the minimum bid is 10^16 wei,
           this should never occur in reality. */
        require(owed <= supply, "supply < claimable");
        return owed;
    }

    function claimableBalanceOf(address account) external view returns (uint) {
        (uint price, uint exercisableDeposits) = market.senderPriceAndExercisableDeposits();
        return _claimableBalanceOf(bidOf[account], price, exercisableDeposits);
    }

    function _totalClaimableSupply(uint exercisableDeposits) internal view returns (uint) {
        uint _totalSupply = totalSupply;
        // We'll avoid throwing an exception here to avoid breaking any dapps, but this case
        // should never occur given the minimum bid size.
        if (exercisableDeposits <= _totalSupply) {
            return 0;
        }
        return exercisableDeposits.sub(_totalSupply);
    }

    function totalClaimableSupply() external view returns (uint) {
        (, uint exercisableDeposits) = market.senderPriceAndExercisableDeposits();
        return _totalClaimableSupply(exercisableDeposits);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function _requireMinimumBid(uint bid) internal pure returns (uint) {
        require(bid >= _MINIMUM_BID || bid == 0, "Balance < $0.01");
        return bid;
    }

    // This must only be invoked during bidding.
    function bid(address bidder, uint newBid) external onlyMarket {
        bidOf[bidder] = _requireMinimumBid(bidOf[bidder].add(newBid));
        totalBids = totalBids.add(newBid);
    }

    // This must only be invoked during bidding.
    function refund(address bidder, uint newRefund) external onlyMarket {
        // The safe subtraction will catch refunds that are too large.
        bidOf[bidder] = _requireMinimumBid(bidOf[bidder].sub(newRefund));
        totalBids = totalBids.sub(newRefund);
    }

    // This must only be invoked after bidding.
    function claim(
        address claimant,
        uint price,
        uint depositsRemaining
    ) external onlyMarket returns (uint optionsClaimed) {
        uint _bid = bidOf[claimant];
        uint claimable = _claimableBalanceOf(_bid, price, depositsRemaining);
        // No options to claim? Nothing happens.
        if (claimable == 0) {
            return 0;
        }

        totalBids = totalBids.sub(_bid);
        bidOf[claimant] = 0;

        totalSupply = totalSupply.add(claimable);
        balanceOf[claimant] = balanceOf[claimant].add(claimable); // Increment rather than assigning since a
transfer may have occurred.

        emit Transfer(address(0), claimant, claimable);
```

```
        emit Issued(claimant, claimable);

        return claimable;
    }

    // This must only be invoked after maturity.
    function exercise(address claimant) external onlyMarket {
        uint balance = balanceOf[claimant];

        if (balance == 0) {
            return;
        }

        balanceOf[claimant] = 0;
        totalSupply = totalSupply.sub(balance);

        emit Transfer(claimant, address(0), balance);
        emit Burned(claimant, balance);
    }

    // This must only be invoked after the exercise window is complete.
    // Note that any options which have not been exercised will linger.
    function expire(address payable beneficiary) external onlyMarket {
        selfdestruct(beneficiary);
    }

    /* ---------- ERC20 Functions ---------- */

    // This should only operate after bidding;
    // Since options can't be claimed until after bidding, all balances are zero until that time.
    // So we don't need to explicitly check the timestamp to prevent transfers.
    function _transfer(
        address _from,
        address _to,
        uint _value
    ) internal returns (bool success) {
        market.requireActiveAndUnpaused();
        require(_to != address(0) && _to != address(this), "Invalid address");

        uint fromBalance = balanceOf[_from];
        require(_value <= fromBalance, "Insufficient balance");

        balanceOf[_from] = fromBalance.sub(_value);
        balanceOf[_to] = balanceOf[_to].add(_value);

        emit Transfer(_from, _to, _value);
        return true;
    }

    function transfer(address _to, uint _value) external returns (bool success) {
        return _transfer(msg.sender, _to, _value);
    }

    function transferFrom(
        address _from,
        address _to,
        uint _value
    ) external returns (bool success) {
        uint fromAllowance = allowance[_from][msg.sender];
        require(_value <= fromAllowance, "Insufficient allowance");

        allowance[_from][msg.sender] = fromAllowance.sub(_value);
        return _transfer(_from, _to, _value);
    }

    function approve(address _spender, uint _value) external returns (bool success) {
        require(_spender != address(0));
        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    /* ========== MODIFIERS ========== */

    modifier onlyMarket() {
        require(msg.sender == address(market), "Only market allowed");
        _;
    }

    /* ========== EVENTS ========== */

    event Issued(address indexed account, uint value);
    event Burned(address indexed account, uint value);
    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);
}
```

BinaryOptionMarket.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./interfaces/IBinaryOptionMarket.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./BinaryOptionMarketManager.sol";
import "./BinaryOption.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/IFeePool.sol";


// https://docs.synthetix.io/contracts/source/contracts/binaryoptionmarket
contract BinaryOptionMarket is Owned, MixinResolver, IBinaryOptionMarket {
    /* ========== LIBRARIES ========== */

    using SafeMath for uint;
    using SafeDecimalMath for uint;

    /* ========== TYPES ========== */

    struct Options {
        BinaryOption long;
        BinaryOption short;
    }

    struct Prices {
        uint long;
        uint short;
    }

    struct Times {
        uint biddingEnd;
        uint maturity;
        uint expiry;
    }

    struct OracleDetails {
        bytes32 key;
        uint strikePrice;
        uint finalPrice;
    }

    /* ========== STATE VARIABLES ========== */

    Options public options;
    Prices public prices;
    Times public times;
    OracleDetails public oracleDetails;
    BinaryOptionMarketManager.Fees public fees;
    BinaryOptionMarketManager.CreatorLimits public creatorLimits;

    // `deposited` tracks the sum of open bids on short and long, plus withheld refund fees.
    // This must explicitly be kept, in case tokens are transferred to the contract directly.
    uint public deposited;
    address public creator;
    bool public resolved;
    bool public refundsEnabled;

    uint internal _feeMultiplier;

    /* ---------- Address Resolver Configuration ---------- */

    bytes32 internal constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 internal constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 internal constant CONTRACT_ZASSETZUSD = "ZassetzUSD";
    bytes32 internal constant CONTRACT_FEEPOOL = "FeePool";

    bytes32[24] internal addressesToCache = [CONTRACT_SYSTEMSTATUS, CONTRACT_EXRATES,
CONTRACT_ZASSETZUSD, CONTRACT_FEEPOOL];

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address _owner,
        address _creator,
        uint[2] memory _creatorLimits, // [capitalRequirement, skewLimit]
        bytes32 _oracleKey,
```

```
        uint _strikePrice,
        bool _refundsEnabled,
        uint[3] memory _times, // [biddingEnd, maturity, expiry]
        uint[2] memory _bids, // [longBid, shortBid]
        uint[3] memory _fees // [poolFee, creatorFee, refundFee]
    )
        public
        Owned(_owner)
        MixinResolver(_owner, addressesToCache) // The resolver is initially set to the owner, but it will be set correctly when the cache is synchronised
    {
        creator = _creator;
        creatorLimits = BinaryOptionMarketManager.CreatorLimits(_creatorLimits[0], _creatorLimits[1]);

        oracleDetails = OracleDetails(_oracleKey, _strikePrice, 0);
        times = Times(_times[0], _times[1], _times[2]);

        refundsEnabled = _refundsEnabled;

        (uint longBid, uint shortBid) = (_bids[0], _bids[1]);
        _checkCreatorLimits(longBid, shortBid);
        emit Bid(Side.Long, _creator, longBid);
        emit Bid(Side.Short, _creator, shortBid);

        // Note that the initial deposit of synths must be made by the manager, otherwise the contract's assumed
        // deposits will fall out of sync with its actual balance. Similarly the total system deposits must be updated in the manager.
        // A balance check isn't performed here since the manager doesn't know the address of the new contract until after it is created.
        uint initialDeposit = longBid.add(shortBid);
        deposited = initialDeposit;

        (uint poolFee, uint creatorFee) = (_fees[0], _fees[1]);
        fees = BinaryOptionMarketManager.Fees(poolFee, creatorFee, _fees[2]);
        _feeMultiplier = SafeDecimalMath.unit().sub(poolFee.add(creatorFee));

        // Compute the prices now that the fees and deposits have been set.
        _updatePrices(longBid, shortBid, initialDeposit);

        // Instantiate the options themselves
        options.long = new BinaryOption(_creator, longBid);
        options.short = new BinaryOption(_creator, shortBid);
    }

    /* ========== VIEWS ========== */

    /* ---------- External Contracts ---------- */

    function _systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus"));
    }

    function _exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates"));
    }

    function _zUSD() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_ZASSETZUSD, "Missing ZassetzUSD"));
    }

    function _feePool() internal view returns (IFeePool) {
        return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL, "Missing FeePool"));
    }

    function _manager() internal view returns (BinaryOptionMarketManager) {
        return BinaryOptionMarketManager(owner);
    }

    /* ---------- Phases ---------- */

    function _biddingEnded() internal view returns (bool) {
        return times.biddingEnd < now;
    }

    function _matured() internal view returns (bool) {
        return times.maturity < now;
    }

    function _expired() internal view returns (bool) {
        return resolved && (times.expiry < now || deposited == 0);
    }

    function phase() external view returns (Phase) {
        if (!_biddingEnded()) {
            return Phase.Bidding;
        }
        if (!_matured()) {
```

```
            return Phase.Trading;
        }
        if (!_expired()) {
            return Phase.Maturity;
        }
        return Phase.Expiry;
    }

    /* ---------- Market Resolution ---------- */

    function _oraclePriceAndTimestamp() internal view returns (uint price, uint updatedAt) {
        return _exchangeRates().rateAndUpdatedTime(oracleDetails.key);
    }

    function oraclePriceAndTimestamp() external view returns (uint price, uint updatedAt) {
        return _oraclePriceAndTimestamp();
    }

    function _isFreshPriceUpdateTime(uint timestamp) internal view returns (bool) {
        (uint maxOraclePriceAge, , ) = _manager().durations();
        return (times.maturity.sub(maxOraclePriceAge)) <= timestamp;
    }

    function canResolve() external view returns (bool) {
        (, uint updatedAt) = _oraclePriceAndTimestamp();
        return !resolved && _matured() && _isFreshPriceUpdateTime(updatedAt);
    }

    function _result() internal view returns (Side) {
        uint price;
        if (resolved) {
            price = oracleDetails.finalPrice;
        } else {
            (price, ) = _oraclePriceAndTimestamp();
        }

        return oracleDetails.strikePrice <= price ? Side.Long : Side.Short;
    }

    function result() external view returns (Side) {
        return _result();
    }

    /* ---------- Option Prices ---------- */

    function _computePrices(
        uint longBids,
        uint shortBids,
        uint _deposited
    ) internal view returns (uint long, uint short) {
        require(longBids != 0 && shortBids != 0, "Bids must be nonzero");
        uint optionsPerSide = _exercisableDeposits(_deposited);

        // The math library rounds up on an exact half-increment -- the price on one side may be an increment too
high,
        // but this only implies a tiny extra quantity will go to fees.
        return                                          (longBids.divideDecimalRound(optionsPerSide),
shortBids.divideDecimalRound(optionsPerSide));
    }

    function senderPriceAndExercisableDeposits() external view returns (uint price, uint exercisable) {
        // When the market is not yet resolved, both sides might be able to exercise all the options.
        // On the other hand, if the market has resolved, then only the winning side may exercise.
        exercisable = 0;
        if (!resolved || address(_option(_result())) == msg.sender) {
            exercisable = _exercisableDeposits(deposited);
        }

        // Send the correct price for each side of the market.
        if (msg.sender == address(options.long)) {
            price = prices.long;
        } else if (msg.sender == address(options.short)) {
            price = prices.short;
        } else {
            revert("Sender is not an option");
        }
    }

    function pricesAfterBidOrRefund(
        Side side,
        uint value,
        bool refund
    ) external view returns (uint long, uint short) {
        (uint longTotalBids, uint shortTotalBids) = _totalBids();
        // prettier-ignore
        function(uint, uint) pure returns (uint) operation = refund ? SafeMath.sub : SafeMath.add;
```

```
        if (side == Side.Long) {
            longTotalBids = operation(longTotalBids, value);
        } else {
            shortTotalBids = operation(shortTotalBids, value);
        }

        if (refund) {
            value = value.multiplyDecimalRound(SafeDecimalMath.unit().sub(fees.refundFee));
        }
        return _computePrices(longTotalBids, shortTotalBids, operation(deposited, value));
    }

    // Returns zero if the result would be negative. See the docs for the formulae this implements.
    function bidOrRefundForPrice(
        Side bidSide,
        Side priceSide,
        uint price,
        bool refund
    ) external view returns (uint) {
        uint adjustedPrice = price.multiplyDecimalRound(_feeMultiplier);
        uint bids = _option(priceSide).totalBids();
        uint _deposited = deposited;
        uint unit = SafeDecimalMath.unit();
        uint refundFeeMultiplier = unit.sub(fees.refundFee);

        if (bidSide == priceSide) {
            uint depositedByPrice = _deposited.multiplyDecimalRound(adjustedPrice);

            // For refunds, the numerator is the negative of the bid case and,
            // in the denominator the adjusted price has an extra factor of (1 - the refundFee).
            if (refund) {
                (depositedByPrice, bids) = (bids, depositedByPrice);
                adjustedPrice = adjustedPrice.multiplyDecimalRound(refundFeeMultiplier);
            }

            // The adjusted price is guaranteed to be less than 1: all its factors are also less than 1.
            return _subToZero(depositedByPrice, bids).divideDecimalRound(unit.sub(adjustedPrice));
        } else {
            uint bidsPerPrice = bids.divideDecimalRound(adjustedPrice);

            // For refunds, the numerator is the negative of the bid case.
            if (refund) {
                (bidsPerPrice, _deposited) = (_deposited, bidsPerPrice);
            }

            uint value = _subToZero(bidsPerPrice, _deposited);
            return refund ? value.divideDecimalRound(refundFeeMultiplier) : value;
        }
    }

    /* ---------- Option Balances and Bids ---------- */

    function _bidsOf(address account) internal view returns (uint long, uint short) {
        return (options.long.bidOf(account), options.short.bidOf(account));
    }

    function bidsOf(address account) external view returns (uint long, uint short) {
        return _bidsOf(account);
    }

    function _totalBids() internal view returns (uint long, uint short) {
        return (options.long.totalBids(), options.short.totalBids());
    }

    function totalBids() external view returns (uint long, uint short) {
        return _totalBids();
    }

    function _claimableBalancesOf(address account) internal view returns (uint long, uint short) {
        return (options.long.claimableBalanceOf(account), options.short.claimableBalanceOf(account));
    }

    function claimableBalancesOf(address account) external view returns (uint long, uint short) {
        return _claimableBalancesOf(account);
    }

    function totalClaimableSupplies() external view returns (uint long, uint short) {
        return (options.long.totalClaimableSupply(), options.short.totalClaimableSupply());
    }

    function _balancesOf(address account) internal view returns (uint long, uint short) {
        return (options.long.balanceOf(account), options.short.balanceOf(account));
    }

    function balancesOf(address account) external view returns (uint long, uint short) {
        return _balancesOf(account);
    }
```

```
    function totalSupplies() external view returns (uint long, uint short) {
        return (options.long.totalSupply(), options.short.totalSupply());
    }

    function _exercisableDeposits(uint _deposited) internal view returns (uint) {
        // Fees are deducted at resolution, so remove them if we're still bidding or trading.
        return resolved ? _deposited : _deposited.multiplyDecimalRound(_feeMultiplier);
    }

    function exercisableDeposits() external view returns (uint) {
        return _exercisableDeposits(deposited);
    }

    /* ---------- Utilities ---------- */

    function _chooseSide(
        Side side,
        uint longValue,
        uint shortValue
    ) internal pure returns (uint) {
        if (side == Side.Long) {
            return longValue;
        }
        return shortValue;
    }

    function _option(Side side) internal view returns (BinaryOption) {
        if (side == Side.Long) {
            return options.long;
        }
        return options.short;
    }

    // Returns zero if the result would be negative.
    function _subToZero(uint a, uint b) internal pure returns (uint) {
        return a < b ? 0 : a.sub(b);
    }

    function _checkCreatorLimits(uint longBid, uint shortBid) internal view {
        uint totalBid = longBid.add(shortBid);
        require(creatorLimits.capitalRequirement <= totalBid, "Insufficient capital");
        uint skewLimit = creatorLimits.skewLimit;
        require(
            skewLimit <= longBid.divideDecimal(totalBid) && skewLimit <=
shortBid.divideDecimal(totalBid),
            "Bids too skewed"
        );
    }

    function _incrementDeposited(uint value) internal returns (uint _deposited) {
        _deposited = deposited.add(value);
        deposited = _deposited;
        _manager().incrementTotalDeposited(value);
    }

    function _decrementDeposited(uint value) internal returns (uint _deposited) {
        _deposited = deposited.sub(value);
        deposited = _deposited;
        _manager().decrementTotalDeposited(value);
    }

    function _requireManagerNotPaused() internal view {
        require(!_manager().paused(), "This action cannot be performed while the contract is paused");
    }

    function requireActiveAndUnpaused() external view {
        _systemStatus().requireSystemActive();
        _requireManagerNotPaused();
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /* ---------- Bidding and Refunding ---------- */

    function _updatePrices(
        uint longBids,
        uint shortBids,
        uint _deposited
    ) internal {
        (uint256 longPrice, uint256 shortPrice) = _computePrices(longBids, shortBids, _deposited);
        prices = Prices(longPrice, shortPrice);
        emit PricesUpdated(longPrice, shortPrice);
    }

    function bid(Side side, uint value) external duringBidding {
        if (value == 0) {
```

```
            return;
        }

        _option(side).bid(msg.sender, value);
        emit Bid(side, msg.sender, value);

        uint _deposited = _incrementDeposited(value);
        _zUSD().transferFrom(msg.sender, address(this), value);

        (uint longTotalBids, uint shortTotalBids) = _totalBids();
        _updatePrices(longTotalBids, shortTotalBids, _deposited);
    }

    function refund(Side side, uint value) external duringBidding returns (uint refundMinusFee) {
        require(refundsEnabled, "Refunds disabled");
        if (value == 0) {
            return 0;
        }

        // Require the market creator to leave sufficient capital in the market.
        if (msg.sender == creator) {
            (uint thisBid, uint thatBid) = _bidsOf(msg.sender);
            if (side == Side.Short) {
                (thisBid, thatBid) = (thatBid, thisBid);
            }
            _checkCreatorLimits(thisBid.sub(value), thatBid);
        }

        // Safe subtraction here and in related contracts will fail if either the
        // total supply, deposits, or wallet balance are too small to support the refund.
        refundMinusFee = value.multiplyDecimalRound(SafeDecimalMath.unit().sub(fees.refundFee));

        _option(side).refund(msg.sender, value);
        emit Refund(side, msg.sender, refundMinusFee, value.sub(refundMinusFee));

        uint _deposited = _decrementDeposited(refundMinusFee);
        _zUSD().transfer(msg.sender, refundMinusFee);

        (uint longTotalBids, uint shortTotalBids) = _totalBids();
        _updatePrices(longTotalBids, shortTotalBids, _deposited);
    }

    /* ---------- Market Resolution ---------- */

    function resolve() external onlyOwner afterMaturity systemActive managerNotPaused {
        require(!resolved, "Market already resolved");

        // We don't need to perform stale price checks, so long as the price was
        // last updated recently enough before the maturity date.
        (uint price, uint updatedAt) = _oraclePriceAndTimestamp();
        require(_isFreshPriceUpdateTime(updatedAt), "Price is stale");

        oracleDetails.finalPrice = price;
        resolved = true;

        // Now remit any collected fees.
        // Since the constructor enforces that creatorFee + poolFee < 1, the balance
        // in the contract will be sufficient to cover these transfers.
        IERC20 zUSD = _zUSD();

        uint _deposited = deposited;
        uint poolFees = _deposited.multiplyDecimalRound(fees.poolFee);
        uint creatorFees = _deposited.multiplyDecimalRound(fees.creatorFee);
        _decrementDeposited(creatorFees.add(poolFees));
        zUSD.transfer(_feePool().FEE_ADDRESS(), poolFees);
        zUSD.transfer(creator, creatorFees);

        emit MarketResolved(_result(), price, updatedAt, deposited, poolFees, creatorFees);
    }

    /* ---------- Claiming and Exercising Options ---------- */

    function _claimOptions()
        internal
        systemActive
        managerNotPaused
        afterBidding
        returns (uint longClaimed, uint shortClaimed)
    {
        uint exercisable = _exercisableDeposits(deposited);
        Side outcome = _result();
        bool _resolved = resolved;

        // Only claim options if we aren't resolved, and only claim the winning side.
        uint longOptions;
        uint shortOptions;
        if (!_resolved || outcome == Side.Long) {
```

```
            longOptions = options.long.claim(msg.sender, prices.long, exercisable);
        }
        if (!_resolved || outcome == Side.Short) {
            shortOptions = options.short.claim(msg.sender, prices.short, exercisable);
        }

        require(longOptions != 0 || shortOptions != 0, "Nothing to claim");
        emit OptionsClaimed(msg.sender, longOptions, shortOptions);
        return (longOptions, shortOptions);
    }

    function claimOptions() external returns (uint longClaimed, uint shortClaimed) {
        return _claimOptions();
    }

    function exerciseOptions() external returns (uint) {
        // The market must be resolved if it has not been.
        if (!resolved) {
            _manager().resolveMarket(address(this));
        }

        // If there are options to be claimed, claim them and proceed.
        (uint claimableLong, uint claimableShort) = _claimableBalancesOf(msg.sender);
        if (claimableLong != 0 || claimableShort != 0) {
            _claimOptions();
        }

        // If the account holds no options, revert.
        (uint longBalance, uint shortBalance) = _balancesOf(msg.sender);
        require(longBalance != 0 || shortBalance != 0, "Nothing to exercise");

        // Each option only needs to be exercised if the account holds any of it.
        if (longBalance != 0) {
            options.long.exercise(msg.sender);
        }
        if (shortBalance != 0) {
            options.short.exercise(msg.sender);
        }

        // Only pay out the side that won.
        uint payout = _chooseSide(_result(), longBalance, shortBalance);
        emit OptionsExercised(msg.sender, payout);
        if (payout != 0) {
            _decrementDeposited(payout);
            _zUSD().transfer(msg.sender, payout);
        }
        return payout;
    }

    /* ---------- Market Expiry ---------- */

    function _selfDestruct(address payable beneficiary) internal {
        uint _deposited = deposited;
        if (_deposited != 0) {
            _decrementDeposited(_deposited);
        }

        // Transfer the balance rather than the deposit value in case there are any synths left over
        // from direct transfers.
        IERC20 zUSD = _zUSD();
        uint balance = zUSD.balanceOf(address(this));
        if (balance != 0) {
            zUSD.transfer(beneficiary, balance);
        }

        // Destroy the option tokens before destroying the market itself.
        options.long.expire(beneficiary);
        options.short.expire(beneficiary);
        selfdestruct(beneficiary);
    }

    function cancel(address payable beneficiary) external onlyOwner duringBidding {
        (uint longTotalBids, uint shortTotalBids) = _totalBids();
        (uint creatorLongBids, uint creatorShortBids) = _bidsOf(creator);
        bool cancellable = longTotalBids == creatorLongBids && shortTotalBids == creatorShortBids;
        require(cancellable, "Not cancellable");
        _selfDestruct(beneficiary);
    }

    function expire(address payable beneficiary) external onlyOwner {
        require(_expired(), "Unexpired options remaining");
        _selfDestruct(beneficiary);
    }

    /* ========== MODIFIERS ========== */

    modifier duringBidding() {
```

```
        require(!_biddingEnded(), "Bidding inactive");
        _;
    }

    modifier afterBidding() {
        require(_biddingEnded(), "Bidding incomplete");
        _;
    }

    modifier afterMaturity() {
        require(_matured(), "Not yet mature");
        _;
    }

    modifier systemActive() {
        _systemStatus().requireSystemActive();
        _;
    }

    modifier managerNotPaused() {
        _requireManagerNotPaused();
        _;
    }

    /* ========== EVENTS ========== */

    event Bid(Side side, address indexed account, uint value);
    event Refund(Side side, address indexed account, uint value, uint fee);
    event PricesUpdated(uint longPrice, uint shortPrice);
    event MarketResolved(
        Side result,
        uint oraclePrice,
        uint oracleTimestamp,
        uint deposited,
        uint poolFees,
        uint creatorFees
    );
    event OptionsClaimed(address indexed account, uint longOptions, uint shortOptions);
    event OptionsExercised(address indexed account, uint value);
}
```

BinaryOptionMarketData.sol

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

// Inheritance
import "./BinaryOption.sol";
import "./BinaryOptionMarket.sol";
import "./BinaryOptionMarketManager.sol";


// https://docs.synthetix.io/contracts/source/contracts/binaryoptionmarketdata
contract BinaryOptionMarketData {
    struct OptionValues {
        uint long;
        uint short;
    }

    struct Deposits {
        uint deposited;
        uint exercisableDeposits;
    }

    struct Resolution {
        bool resolved;
        bool canResolve;
    }

    struct OraclePriceAndTimestamp {
        uint price;
        uint updatedAt;
    }

    // used for things that don't change over the lifetime of the contract
    struct MarketParameters {
        address creator;
        BinaryOptionMarket.Options options;
        BinaryOptionMarket.Times times;
        BinaryOptionMarket.OracleDetails oracleDetails;
        BinaryOptionMarketManager.Fees fees;
        BinaryOptionMarketManager.CreatorLimits creatorLimits;
    }

    struct MarketData {
```

```
        OraclePriceAndTimestamp oraclePriceAndTimestamp;
        BinaryOptionMarket.Prices prices;
        Deposits deposits;
        Resolution resolution;
        BinaryOptionMarket.Phase phase;
        BinaryOptionMarket.Side result;
        OptionValues totalBids;
        OptionValues totalClaimableSupplies;
        OptionValues totalSupplies;
    }

    struct AccountData {
        OptionValues bids;
        OptionValues claimable;
        OptionValues balances;
    }

    function getMarketParameters(BinaryOptionMarket market) public view returns (MarketParameters memory)
{
        (BinaryOption long, BinaryOption short) = market.options();
        (uint biddingEndDate, uint maturityDate, uint expiryDate) = market.times();
        (bytes32 key, uint strikePrice, uint finalPrice) = market.oracleDetails();
        (uint poolFee, uint creatorFee, uint refundFee) = market.fees();

        MarketParameters memory data = MarketParameters(
            market.creator(),
            BinaryOptionMarket.Options(long, short),
            BinaryOptionMarket.Times(biddingEndDate, maturityDate, expiryDate),
            BinaryOptionMarket.OracleDetails(key, strikePrice, finalPrice),
            BinaryOptionMarketManager.Fees(poolFee, creatorFee, refundFee),
            BinaryOptionMarketManager.CreatorLimits(0, 0)
        );

        // Stack too deep otherwise.
        (uint capitalRequirement, uint skewLimit) = market.creatorLimits();
        data.creatorLimits = BinaryOptionMarketManager.CreatorLimits(capitalRequirement, skewLimit);
        return data;
    }

    function getMarketData(BinaryOptionMarket market) public view returns (MarketData memory) {
        (uint price, uint updatedAt) = market.oraclePriceAndTimestamp();
        (uint longClaimable, uint shortClaimable) = market.totalClaimableSupplies();
        (uint longSupply, uint shortSupply) = market.totalSupplies();
        (uint longBids, uint shortBids) = market.totalBids();
        (uint longPrice, uint shortPrice) = market.prices();

        return
            MarketData(
                OraclePriceAndTimestamp(price, updatedAt),
                BinaryOptionMarket.Prices(longPrice, shortPrice),
                Deposits(market.deposited(), market.exercisableDeposits()),
                Resolution(market.resolved(), market.canResolve()),
                market.phase(),
                market.result(),
                OptionValues(longBids, shortBids),
                OptionValues(longClaimable, shortClaimable),
                OptionValues(longSupply, shortSupply)
            );
    }

    function getAccountMarketData(BinaryOptionMarket market, address account) public view returns
(AccountData memory) {
        (uint longBid, uint shortBid) = market.bidsOf(account);
        (uint longClaimable, uint shortClaimable) = market.claimableBalancesOf(account);
        (uint longBalance, uint shortBalance) = market.balancesOf(account);

        return
            AccountData(
                OptionValues(longBid, shortBid),
                OptionValues(longClaimable, shortClaimable),
                OptionValues(longBalance, shortBalance)
            );
    }
}
```

BinaryOptionMarketFactory.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";

// Internal references
import "./BinaryOptionMarket.sol";
```

```
// https://docs.synthetix.io/contracts/source/contracts/binaryoptionmarketfactory
contract BinaryOptionMarketFactory is Owned, MixinResolver {
    /* ========== STATE VARIABLES ========== */

    /* ---------- Address Resolver Configuration ---------- */

    bytes32 internal constant CONTRACT_BINARYOPTIONMARKETMANAGER = "BinaryOptionMarketManager";

    bytes32[24] internal addressesToCache = [CONTRACT_BINARYOPTIONMARKETMANAGER];

    /* ========== CONSTRUCTOR ========== */

    constructor(address _owner, address _resolver) public Owned(_owner) MixinResolver(_resolver, addressesToCache) {}

    /* ========== VIEWS ========== */

    /* ---------- Related Contracts ---------- */

    function _manager() internal view returns (address) {
        return requireAndGetAddress(CONTRACT_BINARYOPTIONMARKETMANAGER, "Missing BinaryOptionMarketManager address");
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function createMarket(
        address creator,
        uint[2] calldata creatorLimits,
        bytes32 oracleKey,
        uint strikePrice,
        bool refundsEnabled,
        uint[3] calldata times, // [biddingEnd, maturity, expiry]
        uint[2] calldata bids, // [longBid, shortBid]
        uint[3] calldata fees // [poolFee, creatorFee, refundFee]
    ) external returns (BinaryOptionMarket) {
        address manager = _manager();
        require(address(manager) == msg.sender, "Only permitted by the manager.");

        return
            new BinaryOptionMarket(
                manager,
                creator,
                creatorLimits,
                oracleKey,
                strikePrice,
                refundsEnabled,
                times,
                bids,
                fees
            );
    }
}
```

BinaryOptionMarketManager.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Pausable.sol";
import "./MixinResolver.sol";
import "./interfaces/IBinaryOptionMarketManager.sol";

// Libraries
import "./AddressSetLib.sol";
import "./SafeDecimalMath.sol";

// Internal references
import "./BinaryOptionMarketFactory.sol";
import "./BinaryOptionMarket.sol";
import "./interfaces/IBinaryOptionMarket.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/binaryoptionmarketmanager
contract BinaryOptionMarketManager is Owned, Pausable, MixinResolver, IBinaryOptionMarketManager {
    /* ========== LIBRARIES ========== */

    using SafeMath for uint;
    using AddressSetLib for AddressSetLib.AddressSet;
```

```
    /* ========== TYPES ========== */

    struct Fees {
        uint poolFee;
        uint creatorFee;
        uint refundFee;
    }

    struct Durations {
        uint maxOraclePriceAge;
        uint expiryDuration;
        uint maxTimeToMaturity;
    }

    struct CreatorLimits {
        uint capitalRequirement;
        uint skewLimit;
    }

    /* ========== STATE VARIABLES ========== */

    Fees public fees;
    Durations public durations;
    CreatorLimits public creatorLimits;

    bool public marketCreationEnabled = true;
    uint public totalDeposited;

    AddressSetLib.AddressSet internal _activeMarkets;
    AddressSetLib.AddressSet internal _maturedMarkets;

    BinaryOptionMarketManager internal _migratingManager;

    /* ---------- Address Resolver Configuration ---------- */

    bytes32 internal constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 internal constant CONTRACT_ZASSETZUSD = "ZassetzUSD";
    bytes32 internal constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 internal constant CONTRACT_BINARYOPTIONMARKETFACTORY =
"BinaryOptionMarketFactory";

    bytes32[24] internal addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_ZASSETZUSD,
        CONTRACT_EXRATES,
        CONTRACT_BINARYOPTIONMARKETFACTORY
    ];

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address _owner,
        address _resolver,
        uint _maxOraclePriceAge,
        uint _expiryDuration,
        uint _maxTimeToMaturity,
        uint _creatorCapitalRequirement,
        uint _creatorSkewLimit,
        uint _poolFee,
        uint _creatorFee,
        uint _refundFee
    ) public Owned(_owner) Pausable() MixinResolver(_resolver, addressesToCache) {
        // Temporarily change the owner so that the setters don't revert.
        owner = msg.sender;
        setExpiryDuration(_expiryDuration);
        setMaxOraclePriceAge(_maxOraclePriceAge);
        setMaxTimeToMaturity(_maxTimeToMaturity);
        setCreatorCapitalRequirement(_creatorCapitalRequirement);
        setCreatorSkewLimit(_creatorSkewLimit);
        setPoolFee(_poolFee);
        setCreatorFee(_creatorFee);
        setRefundFee(_refundFee);
        owner = _owner;
    }

    /* ========== VIEWS ========== */

    /* ---------- Related Contracts ---------- */

    function _systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus
address"));
    }

    function _zUSD() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_ZASSETZUSD, "Missing ZassetzUSD address"));
    }
```

```
    function _exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates"));
    }

    function _factory() internal view returns (BinaryOptionMarketFactory) {
        return
            BinaryOptionMarketFactory(
                requireAndGetAddress(CONTRACT_BINARYOPTIONMARKETFACTORY, "Missing
BinaryOptionMarketFactory address")
            );
    }

    /* ---------- Market Information ---------- */

    function _isKnownMarket(address candidate) internal view returns (bool) {
        return _activeMarkets.contains(candidate) || _maturedMarkets.contains(candidate);
    }

    function numActiveMarkets() external view returns (uint) {
        return _activeMarkets.elements.length;
    }

    function activeMarkets(uint index, uint pageSize) external view returns (address[] memory) {
        return _activeMarkets.getPage(index, pageSize);
    }

    function numMaturedMarkets() external view returns (uint) {
        return _maturedMarkets.elements.length;
    }

    function maturedMarkets(uint index, uint pageSize) external view returns (address[] memory) {
        return _maturedMarkets.getPage(index, pageSize);
    }

    function _isValidKey(bytes32 oracleKey) internal view returns (bool) {
        IExchangeRates exchangeRates = _exchangeRates();

        // If it has a rate, then it's possibly a valid key
        if (exchangeRates.rateForCurrency(oracleKey) != 0) {
            // But not zUSD
            if (oracleKey == "zUSD") {
                return false;
            }

            // and not inverse rates
            (uint entryPoint, , , ) = exchangeRates.inversePricing(oracleKey);
            if (entryPoint != 0) {
                return false;
            }

            return true;
        }

        return false;
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /* ---------- Setters ---------- */

    function setMaxOraclePriceAge(uint _maxOraclePriceAge) public onlyOwner {
        durations.maxOraclePriceAge = _maxOraclePriceAge;
        emit MaxOraclePriceAgeUpdated(_maxOraclePriceAge);
    }

    function setExpiryDuration(uint _expiryDuration) public onlyOwner {
        durations.expiryDuration = _expiryDuration;
        emit ExpiryDurationUpdated(_expiryDuration);
    }

    function setMaxTimeToMaturity(uint _maxTimeToMaturity) public onlyOwner {
        durations.maxTimeToMaturity = _maxTimeToMaturity;
        emit MaxTimeToMaturityUpdated(_maxTimeToMaturity);
    }

    function setPoolFee(uint _poolFee) public onlyOwner {
        uint totalFee = _poolFee + fees.creatorFee;
        require(totalFee < SafeDecimalMath.unit(), "Total fee must be less than 100%.");
        require(0 < totalFee, "Total fee must be nonzero.");
        fees.poolFee = _poolFee;
        emit PoolFeeUpdated(_poolFee);
    }

    function setCreatorFee(uint _creatorFee) public onlyOwner {
        uint totalFee = _creatorFee + fees.poolFee;
        require(totalFee < SafeDecimalMath.unit(), "Total fee must be less than 100%.");
```

```
        require(0 < totalFee, "Total fee must be nonzero.");
        fees.creatorFee = _creatorFee;
        emit CreatorFeeUpdated(_creatorFee);
    }

    function setRefundFee(uint _refundFee) public onlyOwner {
        require(_refundFee <= SafeDecimalMath.unit(), "Refund fee must be no greater than 100%.");
        fees.refundFee = _refundFee;
        emit RefundFeeUpdated(_refundFee);
    }

    function setCreatorCapitalRequirement(uint _creatorCapitalRequirement) public onlyOwner {
        creatorLimits.capitalRequirement = _creatorCapitalRequirement;
        emit CreatorCapitalRequirementUpdated(_creatorCapitalRequirement);
    }

    function setCreatorSkewLimit(uint _creatorSkewLimit) public onlyOwner {
        require(_creatorSkewLimit <= SafeDecimalMath.unit(), "Creator skew limit must be no greater than 1.");
        creatorLimits.skewLimit = _creatorSkewLimit;
        emit CreatorSkewLimitUpdated(_creatorSkewLimit);
    }

    /* ---------- Deposit Management ---------- */

    function incrementTotalDeposited(uint delta) external onlyActiveMarkets notPaused {
        _systemStatus().requireSystemActive();
        totalDeposited = totalDeposited.add(delta);
    }

    function decrementTotalDeposited(uint delta) external onlyKnownMarkets notPaused {
        _systemStatus().requireSystemActive();
        // NOTE: As individual market debt is not tracked here, the underlying markets
        //              need to be careful never to subtract more debt than they added.
        //              This can't be enforced without additional state/communication overhead.
        totalDeposited = totalDeposited.sub(delta);
    }

    /* ---------- Market Lifecycle ---------- */

    function createMarket(
        bytes32 oracleKey,
        uint strikePrice,
        bool refundsEnabled,
        uint[2] calldata times, // [biddingEnd, maturity]
        uint[2] calldata bids // [longBid, shortBid]
    )
        external
        notPaused
        returns (
            IBinaryOptionMarket // no support for returning BinaryOptionMarket polymorphically given the interface
        )
    {
        _systemStatus().requireSystemActive();
        require(marketCreationEnabled, "Market creation is disabled");
        require(_isValidKey(oracleKey), "Invalid key");

        (uint biddingEnd, uint maturity) = (times[0], times[1]);
        require(maturity <= now + durations.maxTimeToMaturity, "Maturity too far in the future");
        uint expiry = maturity.add(durations.expiryDuration);

        uint initialDeposit = bids[0].add(bids[1]);
        require(now < biddingEnd, "End of bidding has passed");
        require(biddingEnd < maturity, "Maturity predates end of bidding");
        // We also require maturity < expiry. But there is no need to check this.
        // Fees being in range are checked in the setters.
        // The market itself validates the capital and skew requirements.

        BinaryOptionMarket market = _factory().createMarket(
            msg.sender,
            [creatorLimits.capitalRequirement, creatorLimits.skewLimit],
            oracleKey,
            strikePrice,
            refundsEnabled,
            [biddingEnd, maturity, expiry],
            bids,
            [fees.poolFee, fees.creatorFee, fees.refundFee]
        );
        market.setResolverAndSyncCache(resolver);
        _activeMarkets.add(address(market));

        // The debt can't be incremented in the new market's constructor because until construction is complete,
        // the manager doesn't know its address in order to grant it permission.
        totalDeposited = totalDeposited.add(initialDeposit);
        _zUSD().transferFrom(msg.sender, address(market), initialDeposit);

        emit MarketCreated(address(market), msg.sender, oracleKey, strikePrice, biddingEnd, maturity, expiry);
```

```
        return market;
    }

    function resolveMarket(address market) external {
        require(_activeMarkets.contains(market), "Not an active market");
        BinaryOptionMarket(market).resolve();
        _activeMarkets.remove(market);
        _maturedMarkets.add(market);
    }

    function cancelMarket(address market) external notPaused {
        require(_activeMarkets.contains(market), "Not an active market");
        address creator = BinaryOptionMarket(market).creator();
        require(msg.sender == creator, "Sender not market creator");
        BinaryOptionMarket(market).cancel(msg.sender);
        _activeMarkets.remove(market);
        emit MarketCancelled(market);
    }

    function expireMarkets(address[] calldata markets) external notPaused {
        for (uint i = 0; i < markets.length; i++) {
            address market = markets[i];

            // The market itself handles decrementing the total deposits.
            BinaryOptionMarket(market).expire(msg.sender);
            // Note that we required that the market is known, which guarantees
            // its index is defined and that the list of markets is not empty.
            _maturedMarkets.remove(market);
            emit MarketExpired(market);
        }
    }

    /* ---------- Upgrade and Administration ---------- */

    function setResolverAndSyncCacheOnMarkets(AddressResolver _resolver, BinaryOptionMarket[] calldata
marketsToSync)
        external
        onlyOwner
    {
        for (uint i = 0; i < marketsToSync.length; i++) {
            marketsToSync[i].setResolverAndSyncCache(_resolver);
        }
    }

    function setMarketCreationEnabled(bool enabled) public onlyOwner {
        if (enabled != marketCreationEnabled) {
            marketCreationEnabled = enabled;
            emit MarketCreationEnabledUpdated(enabled);
        }
    }

    function setMigratingManager(BinaryOptionMarketManager manager) public onlyOwner {
        _migratingManager = manager;
    }

    function migrateMarkets(
        BinaryOptionMarketManager receivingManager,
        bool active,
        BinaryOptionMarket[] calldata marketsToMigrate
    ) external onlyOwner {
        uint _numMarkets = marketsToMigrate.length;
        if (_numMarkets == 0) {
            return;
        }
        AddressSetLib.AddressSet storage markets = active ? _activeMarkets : _maturedMarkets;

        uint runningDepositTotal;
        for (uint i; i < _numMarkets; i++) {
            BinaryOptionMarket market = marketsToMigrate[i];
            require(_isKnownMarket(address(market)), "Market unknown.");

            // Remove it from our list and deposit total.
            markets.remove(address(market));
            runningDepositTotal = runningDepositTotal.add(market.deposited());

            // Prepare to transfer ownership to the new manager.
            market.nominateNewOwner(address(receivingManager));
        }
        // Deduct the total deposits of the migrated markets.
        totalDeposited = totalDeposited.sub(runningDepositTotal);
        emit MarketsMigrated(receivingManager, marketsToMigrate);

        // Now actually transfer the markets over to the new manager.
        receivingManager.receiveMarkets(active, marketsToMigrate);
    }

    function receiveMarkets(bool active, BinaryOptionMarket[] calldata marketsToReceive) external {
```

```
        require(msg.sender == address(_migratingManager), "Only permitted for migrating manager.");

        uint _numMarkets = marketsToReceive.length;
        if (_numMarkets == 0) {
            return;
        }
        AddressSetLib.AddressSet storage markets = active ? _activeMarkets : _maturedMarkets;

        uint runningDepositTotal;
        for (uint i; i < _numMarkets; i++) {
            BinaryOptionMarket market = marketsToReceive[i];
            require(!_isKnownMarket(address(market)), "Market already known.");

            market.acceptOwnership();
            markets.add(address(market));
            // Update the market with the new manager address,
            runningDepositTotal = runningDepositTotal.add(market.deposited());
        }
        totalDeposited = totalDeposited.add(runningDepositTotal);
        emit MarketsReceived(_migratingManager, marketsToReceive);
    }

    /* ========== MODIFIERS ========== */

    modifier onlyActiveMarkets() {
        require(_activeMarkets.contains(msg.sender), "Permitted only for active markets.");
        _;
    }

    modifier onlyKnownMarkets() {
        require(_isKnownMarket(msg.sender), "Permitted only for known markets.");
        _;
    }

    /* ========== EVENTS ========== */

    event MarketCreated(
        address market,
        address indexed creator,
        bytes32 indexed oracleKey,
        uint strikePrice,
        uint biddingEndDate,
        uint maturityDate,
        uint expiryDate
    );
    event MarketExpired(address market);
    event MarketCancelled(address market);
    event MarketsMigrated(BinaryOptionMarketManager receivingManager, BinaryOptionMarket[] markets);
    event MarketsReceived(BinaryOptionMarketManager migratingManager, BinaryOptionMarket[] markets);
    event MarketCreationEnabledUpdated(bool enabled);
    event MaxOraclePriceAgeUpdated(uint duration);
    event ExerciseDurationUpdated(uint duration);
    event ExpiryDurationUpdated(uint duration);
    event MaxTimeToMaturityUpdated(uint duration);
    event CreatorCapitalRequirementUpdated(uint value);
    event CreatorSkewLimitUpdated(uint value);
    event PoolFeeUpdated(uint fee);
    event CreatorFeeUpdated(uint fee);
    event RefundFeeUpdated(uint fee);
}
```

ContractStorage.sol

```
pragma solidity ^0.5.16;

// Internal References
import "./interfaces/IAddressResolver.sol";


// https://docs.synthetix.io/contracts/source/contracts/contractstorage
contract ContractStorage {
    IAddressResolver public resolverProxy;

    mapping(bytes32 => bytes32) public hashes;

    constructor(address _resolver) internal {
        // ReadProxyAddressResolver
        resolverProxy = IAddressResolver(_resolver);
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _memoizeHash(bytes32 contractName) internal returns (bytes32) {
        bytes32 hashKey = hashes[contractName];
        if (hashKey == bytes32(0)) {
            // set to unique hash at the time of creation
```

```
            hashKey = keccak256(abi.encodePacked(msg.sender, contractName, block.number));
            hashes[contractName] = hashKey;
        }
        return hashKey;
    }

    /* ========== VIEWS ========== */

    /* ========== RESTRICTED FUNCTIONS ========== */

    function migrateContractKey(
        bytes32 fromContractName,
        bytes32 toContractName,
        bool removeAccessFromPreviousContract
    ) external onlyContract(fromContractName) {
        require(hashes[fromContractName] != bytes32(0), "Cannot migrate empty contract");

        hashes[toContractName] = hashes[fromContractName];

        if (removeAccessFromPreviousContract) {
            delete hashes[fromContractName];
        }

        emit KeyMigrated(fromContractName, toContractName, removeAccessFromPreviousContract);
    }

    /* ========== MODIFIERS ========== */

    modifier onlyContract(bytes32 contractName) {
        address callingContract = resolverProxy.requireAndGetAddress(
            contractName,
            "Cannot find contract in Address Resolver"
        );
        require(callingContract == msg.sender, "Can only be invoked by the configured contract");
        _;
    }

    /* ========== EVENTS ========== */

    event KeyMigrated(bytes32 fromContractName, bytes32 toContractName, bool
removeAccessFromPreviousContract);
}
```

DappMaintenance.sol

```
pragma solidity ^0.5.16;

import "./Owned.sol";


// https://docs.synthetix.io/contracts/source/contracts/dappmaintenance

/**
 * @title DappMaintenance contract.
 * @dev When the Synthetix system is on maintenance (upgrade, release...etc) the dApps also need
 * to be put on maintenance so no transactions can be done. The DappMaintenance contract is here to keep a state
 of
 * the dApps which indicates if yes or no, they should be up or down.
 */
contract DappMaintenance is Owned {
    bool public isPausedStaking = false;
    bool public isPausedSX = false;

    /**
     * @dev Constructor
     */
    constructor(address _owner) public Owned(_owner) {
        require(_owner != address(0), "Owner address cannot be 0");
        owner = _owner;
        emit OwnerChanged(address(0), _owner);
    }

    function setMaintenanceModeAll(bool isPaused) external onlyOwner {
        isPausedStaking = isPaused;
        isPausedSX = isPaused;
        emit StakingMaintenance(isPaused);
        emit SXMaintenance(isPaused);
    }

    function setMaintenanceModeStaking(bool isPaused) external onlyOwner {
        isPausedStaking = isPaused;
        emit StakingMaintenance(isPausedStaking);
    }

    function setMaintenanceModeSX(bool isPaused) external onlyOwner {
        isPausedSX = isPaused;
```

```
        emit SXMaintenance(isPausedSX);
    }

    event StakingMaintenance(bool isPaused);
    event SXMaintenance(bool isPaused);
}
```

DebtCache.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/IDebtCache.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IIssuer.sol";
import "./interfaces/IExchanger.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IEtherCollateral.sol";
import "./interfaces/IEtherCollateralsUSD.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/debtcache
contract DebtCache is Owned, MixinResolver, MixinSystemSettings, IDebtCache {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    uint internal _cachedDebt;
    mapping(bytes32 => uint) internal _cachedSynthDebt;
    uint internal _cacheTimestamp;
    bool internal _cacheInvalid = true;

    /* ========== ENCODED NAMES ========== */

    bytes32 internal constant zUSD = "zUSD";
    bytes32 internal constant zBNB = "zBNB";

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_ETHERCOLLATERAL = "EtherCollateral";
    bytes32 private constant CONTRACT_ETHERCOLLATERAL_SUSD = "EtherCollateralsUSD";

    bytes32[24] private addressesToCache = [
        CONTRACT_ISSUER,
        CONTRACT_EXCHANGER,
        CONTRACT_EXRATES,
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_ETHERCOLLATERAL,
        CONTRACT_ETHERCOLLATERAL_SUSD
    ];

    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {}

    /* ========== VIEWS ========== */

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    function systemStatus() internal view returns (ISystemStatus) {
```

```
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function etherCollateral() internal view returns (IEtherCollateral) {
        return IEtherCollateral(requireAndGetAddress(CONTRACT_ETHERCOLLATERAL, "Missing EtherCollateral address"));
    }

    function etherCollateralsUSD() internal view returns (IEtherCollateralsUSD) {
        return
            IEtherCollateralsUSD(requireAndGetAddress(CONTRACT_ETHERCOLLATERAL_SUSD, "Missing EtherCollateralsUSD address"));
    }

    function debtSnapshotStaleTime() external view returns (uint) {
        return getDebtSnapshotStaleTime();
    }

    function cachedDebt() external view returns (uint) {
        return _cachedDebt;
    }

    function cachedSynthDebt(bytes32 currencyKey) external view returns (uint) {
        return _cachedSynthDebt[currencyKey];
    }

    function cacheTimestamp() external view returns (uint) {
        return _cacheTimestamp;
    }

    function cacheInvalid() external view returns (bool) {
        return _cacheInvalid;
    }

    function _cacheStale(uint timestamp) internal view returns (bool) {
        // Note a 0 timestamp means that the cache is uninitialised.
        // We'll keep the check explicitly in case the stale time is
        // ever set to something higher than the current unix time (e.g. to turn off staleness).
        return getDebtSnapshotStaleTime() < block.timestamp - timestamp || timestamp == 0;
    }

    function cacheStale() external view returns (bool) {
        return _cacheStale(_cacheTimestamp);
    }

    function _issuedSynthValues(bytes32[] memory currencyKeys, uint[] memory rates) internal view returns (uint[] memory) {
        uint numValues = currencyKeys.length;
        uint[] memory values = new uint[](numValues);
        ISynth[] memory synths = issuer().getSynths(currencyKeys);

        for (uint i = 0; i < numValues; i++) {
            bytes32 key = currencyKeys[i];
            address synthAddress = address(synths[i]);
            require(synthAddress != address(0), "Zasset does not exist");
            uint supply = IERC20(synthAddress).totalSupply();

            bool iszUSD = key == zUSD;
            if (iszUSD || key == zBNB) {
                IEtherCollateral etherCollateralContract = iszUSD
                    ? IEtherCollateral(address(etherCollateralsUSD()))
                    : etherCollateral();
                uint etherCollateralSupply = etherCollateralContract.totalIssuedSynths();
                supply = supply.sub(etherCollateralSupply);
            }

            values[i] = supply.multiplyDecimalRound(rates[i]);
        }
        return values;
    }

    function _currentSynthDebts(bytes32[] memory currencyKeys)
        internal
        view
        returns (uint[] memory snxIssuedDebts, bool anyRateIsInvalid)
    {
        (uint[] memory rates, bool isInvalid) = exchangeRates().ratesAndInvalidForCurrencies(currencyKeys);
        return (_issuedSynthValues(currencyKeys, rates), isInvalid);
    }

    function currentSynthDebts(bytes32[] calldata currencyKeys)
        external
        view
        returns (uint[] memory debtValues, bool anyRateIsInvalid)
    {
        return _currentSynthDebts(currencyKeys);
```

```
    }

    function _cachedSynthDebts(bytes32[] memory currencyKeys) internal view returns (uint[] memory) {
        uint numKeys = currencyKeys.length;
        uint[] memory debts = new uint[](numKeys);
        for (uint i = 0; i < numKeys; i++) {
            debts[i] = _cachedSynthDebt[currencyKeys[i]];
        }
        return debts;
    }

    function cachedSynthDebts(bytes32[] calldata currencyKeys) external view returns (uint[] memory
snxIssuedDebts) {
        return _cachedSynthDebts(currencyKeys);
    }

    function _currentDebt() internal view returns (uint debt, bool anyRateIsInvalid) {
        (uint[] memory values, bool isInvalid) = _currentSynthDebts(issuer().availableCurrencyKeys());
        uint numValues = values.length;
        uint total;
        for (uint i; i < numValues; i++) {
            total = total.add(values[i]);
        }
        return (total, isInvalid);
    }

    function currentDebt() external view returns (uint debt, bool anyRateIsInvalid) {
        return _currentDebt();
    }

    function cacheInfo()
        external
        view
        returns (
            uint debt,
            uint timestamp,
            bool isInvalid,
            bool isStale
        )
    {
        uint time = _cacheTimestamp;
        return (_cachedDebt, time, _cacheInvalid, _cacheStale(time));
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    // This function exists in case a synth is ever somehow removed without its snapshot being updated.
    function purgeCachedSynthDebt(bytes32 currencyKey) external onlyOwner {
        require(issuer().synths(currencyKey) == ISynth(0), "Zasset exists");
        delete _cachedSynthDebt[currencyKey];
    }

    function takeDebtSnapshot() external requireSystemActiveIfNotOwner {
        bytes32[] memory currencyKeys = issuer().availableCurrencyKeys();
        (uint[] memory values, bool isInvalid) = _currentSynthDebts(currencyKeys);

        uint numValues = values.length;
        uint snxCollateralDebt;
        for (uint i; i < numValues; i++) {
            uint value = values[i];
            snxCollateralDebt = snxCollateralDebt.add(value);
            _cachedSynthDebt[currencyKeys[i]] = value;
        }
        _cachedDebt = snxCollateralDebt;
        _cacheTimestamp = block.timestamp;
        emit DebtCacheUpdated(snxCollateralDebt);
        emit DebtCacheSnapshotTaken(block.timestamp);

        // (in)validate the cache if necessary
        _updateDebtCacheValidity(isInvalid);
    }

    function updateCachedSynthDebts(bytes32[] calldata currencyKeys) external requireSystemActiveIfNotOwner
{
        (uint[] memory rates, bool anyRateInvalid) =
exchangeRates().ratesAndInvalidForCurrencies(currencyKeys);
        _updateCachedSynthDebtsWithRates(currencyKeys, rates, anyRateInvalid);
    }

    function updateCachedSynthDebtWithRate(bytes32 currencyKey, uint currencyRate) external onlyIssuer {
        bytes32[] memory synthKeyArray = new bytes32[](1);
        synthKeyArray[0] = currencyKey;
        uint[] memory synthRateArray = new uint[](1);
        synthRateArray[0] = currencyRate;
        _updateCachedSynthDebtsWithRates(synthKeyArray, synthRateArray, false);
    }
```

```
    function updateCachedSynthDebtsWithRates(bytes32[] calldata currencyKeys, uint[] calldata currencyRates)
        external
        onlyIssuerOrExchanger
    {
        _updateCachedSynthDebtsWithRates(currencyKeys, currencyRates, false);
    }

    function updateDebtCacheValidity(bool currentlyInvalid) external onlyIssuer {
        _updateDebtCacheValidity(currentlyInvalid);
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _updateDebtCacheValidity(bool currentlyInvalid) internal {
        if (_cacheInvalid != currentlyInvalid) {
            _cacheInvalid = currentlyInvalid;
            emit DebtCacheValidityChanged(currentlyInvalid);
        }
    }

    function _updateCachedSynthDebtsWithRates(
        bytes32[] memory currencyKeys,
        uint[] memory currentRates,
        bool anyRateIsInvalid
    ) internal {
        uint numKeys = currencyKeys.length;
        require(numKeys == currentRates.length, "Input array lengths differ");

        // Update the cached values for each synth, saving the sums as we go.
        uint cachedSum;
        uint currentSum;
        uint[] memory currentValues = _issuedSynthValues(currencyKeys, currentRates);
        for (uint i = 0; i < numKeys; i++) {
            bytes32 key = currencyKeys[i];
            uint currentSynthDebt = currentValues[i];
            cachedSum = cachedSum.add(_cachedSynthDebt[key]);
            currentSum = currentSum.add(currentSynthDebt);
            _cachedSynthDebt[key] = currentSynthDebt;
        }

        // Compute the difference and apply it to the snapshot
        if (cachedSum != currentSum) {
            uint debt = _cachedDebt;
            // This requirement should never fail, as the total debt snapshot is the sum of the individual synth
            // debt snapshots.
            require(cachedSum <= debt, "Cached zasset sum exceeds total debt");
            debt = debt.sub(cachedSum).add(currentSum);
            _cachedDebt = debt;
            emit DebtCacheUpdated(debt);
        }

        // A partial update can invalidate the debt cache, but a full snapshot must be performed in order
        // to re-validate it.
        if (anyRateIsInvalid) {
            _updateDebtCacheValidity(anyRateIsInvalid);
        }
    }

    /* ========== MODIFIERS ========== */

    function _requireSystemActiveIfNotOwner() internal view {
        if (msg.sender != owner) {
            systemStatus().requireSystemActive();
        }
    }

    modifier requireSystemActiveIfNotOwner() {
        _requireSystemActiveIfNotOwner();
        _;
    }

    function _onlyIssuer() internal view {
        require(msg.sender == address(issuer()), "Sender is not Issuer");
    }

    modifier onlyIssuer() {
        _onlyIssuer();
        _;
    }

    function _onlyIssuerOrExchanger() internal view {
        require(msg.sender == address(issuer()) || msg.sender == address(exchanger()), "Sender is not Issuer or
Exchanger");
    }

    modifier onlyIssuerOrExchanger() {
        _onlyIssuerOrExchanger();
```

```
        _;
    }

    /* ========== EVENTS ========== */

    event DebtCacheUpdated(uint cachedDebt);
    event DebtCacheSnapshotTaken(uint timestamp);
    event DebtCacheValidityChanged(bool indexed isInvalid);
}
```

DelegateApprovals.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./interfaces/IDelegateApprovals.sol";

// Internal references
import "./EternalStorage.sol";


// https://docs.synthetix.io/contracts/source/contracts/delegateapprovals
contract DelegateApprovals is Owned, IDelegateApprovals {
    bytes32 public constant BURN_FOR_ADDRESS = "BurnForAddress";
    bytes32 public constant ISSUE_FOR_ADDRESS = "IssueForAddress";
    bytes32 public constant CLAIM_FOR_ADDRESS = "ClaimForAddress";
    bytes32 public constant EXCHANGE_FOR_ADDRESS = "ExchangeForAddress";
    bytes32 public constant APPROVE_ALL = "ApproveAll";

    bytes32[5] private _delegatableFunctions = [
        APPROVE_ALL,
        BURN_FOR_ADDRESS,
        ISSUE_FOR_ADDRESS,
        CLAIM_FOR_ADDRESS,
        EXCHANGE_FOR_ADDRESS
    ];

    /* ========== STATE VARIABLES ========== */
    EternalStorage public eternalStorage;

    constructor(address _owner, EternalStorage _eternalStorage) public Owned(_owner) {
        eternalStorage = _eternalStorage;
    }

    /* ========== VIEWS ========== */

    // Move it to setter and associatedState

    // util to get key based on action name + address of authoriser + address for delegate
    function _getKey(
        bytes32 _action,
        address _authoriser,
        address _delegate
    ) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked(_action, _authoriser, _delegate));
    }

    // hash of actionName + address of authoriser + address for the delegate
    function canBurnFor(address authoriser, address delegate) external view returns (bool) {
        return _checkApproval(BURN_FOR_ADDRESS, authoriser, delegate);
    }

    function canIssueFor(address authoriser, address delegate) external view returns (bool) {
        return _checkApproval(ISSUE_FOR_ADDRESS, authoriser, delegate);
    }

    function canClaimFor(address authoriser, address delegate) external view returns (bool) {
        return _checkApproval(CLAIM_FOR_ADDRESS, authoriser, delegate);
    }

    function canExchangeFor(address authoriser, address delegate) external view returns (bool) {
        return _checkApproval(EXCHANGE_FOR_ADDRESS, authoriser, delegate);
    }

    function approvedAll(address authoriser, address delegate) public view returns (bool) {
        return eternalStorage.getBooleanValue(_getKey(APPROVE_ALL, authoriser, delegate));
    }

    // internal function to check approval based on action
    // if approved for all actions then will return true
    // before checking specific approvals
    function _checkApproval(
        bytes32 action,
        address authoriser,
```

```
        address delegate
    ) internal view returns (bool) {
        if (approvedAll(authoriser, delegate)) return true;

        return eternalStorage.getBooleanValue(_getKey(action, authoriser, delegate));
    }

    /* ========== SETTERS ========== */

    // Approve All
    function approveAllDelegatePowers(address delegate) external {
        _setApproval(APPROVE_ALL, msg.sender, delegate);
    }

    // Removes all delegate approvals
    function removeAllDelegatePowers(address delegate) external {
        for (uint i = 0; i < _delegatableFunctions.length; i++) {
            _withdrawApproval(_delegatableFunctions[i], msg.sender, delegate);
        }
    }

    // Burn on behalf
    function approveBurnOnBehalf(address delegate) external {
        _setApproval(BURN_FOR_ADDRESS, msg.sender, delegate);
    }

    function removeBurnOnBehalf(address delegate) external {
        _withdrawApproval(BURN_FOR_ADDRESS, msg.sender, delegate);
    }

    // Issue on behalf
    function approveIssueOnBehalf(address delegate) external {
        _setApproval(ISSUE_FOR_ADDRESS, msg.sender, delegate);
    }

    function removeIssueOnBehalf(address delegate) external {
        _withdrawApproval(ISSUE_FOR_ADDRESS, msg.sender, delegate);
    }

    // Claim on behalf
    function approveClaimOnBehalf(address delegate) external {
        _setApproval(CLAIM_FOR_ADDRESS, msg.sender, delegate);
    }

    function removeClaimOnBehalf(address delegate) external {
        _withdrawApproval(CLAIM_FOR_ADDRESS, msg.sender, delegate);
    }

    // Exchange on behalf
    function approveExchangeOnBehalf(address delegate) external {
        _setApproval(EXCHANGE_FOR_ADDRESS, msg.sender, delegate);
    }

    function removeExchangeOnBehalf(address delegate) external {
        _withdrawApproval(EXCHANGE_FOR_ADDRESS, msg.sender, delegate);
    }

    function _setApproval(
        bytes32 action,
        address authoriser,
        address delegate
    ) internal {
        require(delegate != address(0), "Can't delegate to address(0)");
        eternalStorage.setBooleanValue(_getKey(action, authoriser, delegate), true);
        emit Approval(authoriser, delegate, action);
    }

    function _withdrawApproval(
        bytes32 action,
        address authoriser,
        address delegate
    ) internal {
        // Check approval is set otherwise skip deleting approval
        if (eternalStorage.getBooleanValue(_getKey(action, authoriser, delegate))) {
            eternalStorage.deleteBooleanValue(_getKey(action, authoriser, delegate));
            emit WithdrawApproval(authoriser, delegate, action);
        }
    }

    function setEternalStorage(EternalStorage _eternalStorage) external onlyOwner {
        require(address(_eternalStorage) != address(0), "Can't set eternalStorage to address(0)");
        eternalStorage = _eternalStorage;
        emit EternalStorageUpdated(address(eternalStorage));
    }

    /* ========== EVENTS ========== */
    event Approval(address indexed authoriser, address delegate, bytes32 action);
```

```
        event WithdrawApproval(address indexed authoriser, address delegate, bytes32 action);
        event EternalStorageUpdated(address newEternalStorage);
}
```

Depot.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Pausable.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";
import "./MixinResolver.sol";
import "./interfaces/IDepot.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IERC20.sol";
import "./interfaces/IExchangeRates.sol";


// https://docs.synthetix.io/contracts/source/contracts/depot
contract Depot is Owned, Pausable, ReentrancyGuard, MixinResolver, IDepot {
        using SafeMath for uint;
        using SafeDecimalMath for uint;

        bytes32 internal constant HZN = "HZN";
        bytes32 internal constant BNB = "BNB";

        /* ========== STATE VARIABLES ========== */

        // Address where the ether and Synths raised for selling HZN is transfered to
        // Any ether raised for selling Synths gets sent back to whoever deposited the Synths,
        // and doesn't have anything to do with this address.
        address payable public fundsWallet;

        /* Stores deposits from users. */
        struct SynthDepositEntry {
                // The user that made the deposit
                address payable user;
                // The amount (in Zassets) that they deposited
                uint amount;
        }

        /* User deposits are sold on a FIFO (First in First out) basis. When users deposit
             zassets with us, they get added this queue, which then gets fulfilled in order.
             Conceptually this fits well in an array, but then when users fill an order we
             end up copying the whole array around, so better to use an index mapping instead
             for gas performance reasons.

             The indexes are specified (inclusive, exclusive), so (0, 0) means there's nothing
             in the array, and (3, 6) means there are 3 elements at 3, 4, and 5. You can obtain
             the length of the "array" by querying depositEndIndex - depositStartIndex. All index
             operations use safeAdd, so there is no way to overflow, so that means there is a
             very large but finite amount of deposits this contract can handle before it fills up. */
        mapping(uint => SynthDepositEntry) public deposits;
        // The starting index of our queue inclusive
        uint public depositStartIndex;
        // The ending index of our queue exclusive
        uint public depositEndIndex;

        /* This is a convenience variable so users and dApps can just query how much zUSD
             we have available for purchase without having to iterate the mapping with a
             O(n) amount of calls for something we'll probably want to display quite regularly. */
        uint public totalSellableDeposits;

        // The minimum amount of zUSD required to enter the FiFo queue
        uint public minimumDepositAmount = 50 * SafeDecimalMath.unit();

        // A cap on the amount of zUSD you can buy with BNB in 1 transaction
        uint public maxEthPurchase = 500 * SafeDecimalMath.unit();

        // If a user deposits a zasset amount < the minimumDepositAmount the contract will keep
        // the total of small deposits which will not be sold on market and the sender
        // must call withdrawMyDepositedSynths() to get them back.
        mapping(address => uint) public smallDeposits;

        /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

        bytes32 private constant CONTRACT_ZASSETZUSD = "ZassetzUSD";
        bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";
        bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";

        bytes32[24] private addressesToCache = [CONTRACT_ZASSETZUSD, CONTRACT_EXRATES,
```

```
CONTRACT_SYNTHETIX];

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address _owner,
        address payable _fundsWallet,
        address _resolver
    ) public Owned(_owner) Pausable() MixinResolver(_resolver, addressesToCache) {
        fundsWallet = _fundsWallet;
    }

    /* ========== SETTERS ========== */

    function setMaxEthPurchase(uint _maxEthPurchase) external onlyOwner {
        maxEthPurchase = _maxEthPurchase;
        emit MaxEthPurchaseUpdated(maxEthPurchase);
    }

    /**
     * @notice Set the funds wallet where BNB raised is held
     * @param _fundsWallet The new address to forward BNB and Zassets to
     */
    function setFundsWallet(address payable _fundsWallet) external onlyOwner {
        fundsWallet = _fundsWallet;
        emit FundsWalletUpdated(fundsWallet);
    }

    /**
     * @notice Set the minimum deposit amount required to depoist zUSD into the FIFO queue
     * @param _amount The new new minimum number of zUSD required to deposit
     */
    function setMinimumDepositAmount(uint _amount) external onlyOwner {
        // Do not allow us to set it less than 1 dollar opening up to fractional desposits in the queue again
        require(_amount > SafeDecimalMath.unit(), "Minimum deposit amount must be greater than UNIT");
        minimumDepositAmount = _amount;
        emit MinimumDepositAmountUpdated(minimumDepositAmount);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /**
     * @notice Fallback function (exchanges BNB to zUSD)
     */
    function() external payable nonReentrant rateNotInvalid(BNB) notPaused {
        _exchangeEtherForSynths();
    }

    /**
     * @notice Exchange BNB to zUSD.
     */
    /* solhint-disable multiple-sends, reentrancy */
    function exchangeEtherForSynths()
        external
        payable
        nonReentrant
        rateNotInvalid(BNB)
        notPaused
        returns (
            uint // Returns the number of Zassets (zUSD) received
        )
    {
        return _exchangeEtherForSynths();
    }

    function _exchangeEtherForSynths() internal returns (uint) {
        require(msg.value <= maxEthPurchase, "BNB amount above maxEthPurchase limit");
        uint ethToSend;

        // The multiplication works here because exchangeRates().rateForCurrency(BNB) is specified in
        // 18 decimal places, just like our currency base.
        uint requestedToPurchase = msg.value.multiplyDecimal(exchangeRates().rateForCurrency(BNB));
        uint remainingToFulfill = requestedToPurchase;

        // Iterate through our outstanding deposits and sell them one at a time.
        for (uint i = depositStartIndex; remainingToFulfill > 0 && i < depositEndIndex; i++) {
            SynthDepositEntry memory deposit = deposits[i];

            // If it's an empty spot in the queue from a previous withdrawal, just skip over it and
            // update the queue. It's already been deleted.
            if (deposit.user == address(0)) {
                depositStartIndex = depositStartIndex.add(1);
            } else {
                // If the deposit can more than fill the order, we can do this
                // without touching the structure of our queue.
                if (deposit.amount > remainingToFulfill) {
                    // Ok, this deposit can fulfill the whole remainder. We don't need
```

```
                        // to change anything about our queue we can just fulfill it.
                        // Subtract the amount from our deposit and total.
                        uint newAmount = deposit.amount.sub(remainingToFulfill);
                        deposits[i] = SynthDepositEntry({user: deposit.user, amount: newAmount});

                        totalSellableDeposits = totalSellableDeposits.sub(remainingToFulfill);

                        // Transfer the BNB to the depositor. Send is used instead of transfer
                        // so a non payable contract won't block the FIFO queue on a failed
                        // BNB payable for zassets transaction. The proceeds to be sent to the
                        // synthetix foundation funds wallet. This is to protect all depositors
                        // in the queue in this rare case that may occur.
                        ethToSend                                                                    =
remainingToFulfill.divideDecimal(exchangeRates().rateForCurrency(BNB));

                        // We need to use send here instead of transfer because transfer reverts
                        // if the recipient is a non-payable contract. Send will just tell us it
                        // failed by returning false at which point we can continue.
                        if (!deposit.user.send(ethToSend)) {
                            fundsWallet.transfer(ethToSend);
                            emit NonPayableContract(deposit.user, ethToSend);
                        } else {
                            emit ClearedDeposit(msg.sender, deposit.user, ethToSend, remainingToFulfill, i);
                        }

                        // And the Zassets to the recipient.
                        // Note: Fees are calculated by the Zasset contract, so when
                        //           we request a specific transfer here, the fee is
                        //           automatically deducted and sent to the fee pool.
                        synthsUSD().transfer(msg.sender, remainingToFulfill);

                        // And we have nothing left to fulfill on this order.
                        remainingToFulfill = 0;
                    } else if (deposit.amount <= remainingToFulfill) {
                        // We need to fulfill this one in its entirety and kick it out of the queue.
                        // Start by kicking it out of the queue.
                        // Free the storage because we can.
                        delete deposits[i];
                        // Bump our start index forward one.
                        depositStartIndex = depositStartIndex.add(1);
                        // We also need to tell our total it's decreased
                        totalSellableDeposits = totalSellableDeposits.sub(deposit.amount);

                        // Now fulfill by transfering the BNB to the depositor. Send is used instead of transfer
                        // so a non payable contract won't block the FIFO queue on a failed
                        // BNB payable for zassets transaction. The proceeds to be sent to the
                        // synthetix foundation funds wallet. This is to protect all depositors
                        // in the queue in this rare case that may occur.
                        ethToSend = deposit.amount.divideDecimal(exchangeRates().rateForCurrency(BNB));

                        // We need to use send here instead of transfer because transfer reverts
                        // if the recipient is a non-payable contract. Send will just tell us it
                        // failed by returning false at which point we can continue.
                        if (!deposit.user.send(ethToSend)) {
                            fundsWallet.transfer(ethToSend);
                            emit NonPayableContract(deposit.user, ethToSend);
                        } else {
                            emit ClearedDeposit(msg.sender, deposit.user, ethToSend, deposit.amount, i);
                        }

                        // And the Zassets to the recipient.
                        // Note: Fees are calculated by the Zasset contract, so when
                        //           we request a specific transfer here, the fee is
                        //           automatically deducted and sent to the fee pool.
                        synthsUSD().transfer(msg.sender, deposit.amount);

                        // And subtract the order from our outstanding amount remaining
                        // for the next iteration of the loop.
                        remainingToFulfill = remainingToFulfill.sub(deposit.amount);
                    }
                }
            }

            // Ok, if we're here and 'remainingToFulfill' isn't zero, then
            // we need to refund the remainder of their BNB back to them.
            if (remainingToFulfill > 0) {
                msg.sender.transfer(remainingToFulfill.divideDecimal(exchangeRates().rateForCurrency(BNB)));
            }

            // How many did we actually give them?
            uint fulfilled = requestedToPurchase.sub(remainingToFulfill);

            if (fulfilled > 0) {
                // Now tell everyone that we gave them that many (only if the amount is greater than 0).
                emit Exchange("BNB", msg.value, "zUSD", fulfilled);
            }
```

```
        return fulfilled;
    }

    /* solhint-enable multiple-sends, reentrancy */

    /**
     * @notice Exchange BNB to zUSD while insisting on a particular rate. This allows a user to
     *         exchange while protecting against frontrunning by the contract owner on the exchange rate.
     * @param guaranteedRate The exchange rate (ether price) which must be honored or the call will revert.
     */
    function exchangeEtherForSynthsAtRate(uint guaranteedRate)
        external
        payable
        rateNotInvalid(BNB)
        notPaused
        returns (
            uint // Returns the number of Zassets (zUSD) received
        )
    {
        require(guaranteedRate == exchangeRates().rateForCurrency(BNB), "Guaranteed rate would not be received");

        return _exchangeEtherForSynths();
    }

    function _exchangeEtherForSNX() internal returns (uint) {
        // How many HZN are they going to be receiving?
        uint synthetixToSend = synthetixReceivedForEther(msg.value);

        // Store the BNB in our funds wallet
        fundsWallet.transfer(msg.value);

        // And send them the HZN.
        synthetix().transfer(msg.sender, synthetixToSend);

        emit Exchange("BNB", msg.value, "HZN", synthetixToSend);

        return synthetixToSend;
    }

    /**
     * @notice Exchange BNB to HZN.
     */
    function exchangeEtherForSNX()
        external
        payable
        rateNotInvalid(HZN)
        rateNotInvalid(BNB)
        notPaused
        returns (
            uint // Returns the number of HZN received
        )
    {
        return _exchangeEtherForSNX();
    }

    /**
     * @notice Exchange BNB to HZN while insisting on a particular set of rates. This allows a user to
     *         exchange while protecting against frontrunning by the contract owner on the exchange rates.
     * @param guaranteedEtherRate The ether exchange rate which must be honored or the call will revert.
     * @param guaranteedSynthetixRate The synthetix exchange rate which must be honored or the call will revert.
     */
    function exchangeEtherForSNXAtRate(uint guaranteedEtherRate, uint guaranteedSynthetixRate)
        external
        payable
        rateNotInvalid(HZN)
        rateNotInvalid(BNB)
        notPaused
        returns (
            uint // Returns the number of HZN received
        )
    {
        require(guaranteedEtherRate == exchangeRates().rateForCurrency(BNB), "Guaranteed ether rate would not be received");
        require(
            guaranteedSynthetixRate == exchangeRates().rateForCurrency(HZN),
            "Guaranteed synthetix rate would not be received"
        );

        return _exchangeEtherForSNX();
    }

    function _exchangeSynthsForSNX(uint synthAmount) internal returns (uint) {
        // How many HZN are they going to be receiving?
        uint synthetixToSend = synthetixReceivedForSynths(synthAmount);

        // Ok, transfer the Synths to our funds wallet.
```

```
        // These do not go in the deposit queue as they aren't for sale as such unless
        // they're sent back in from the funds wallet.
        synthsUSD().transferFrom(msg.sender, fundsWallet, synthAmount);

        // And send them the HZN.
        synthetix().transfer(msg.sender, synthetixToSend);

        emit Exchange("zUSD", synthAmount, "HZN", synthetixToSend);

        return synthetixToSend;
    }

    /**
     * @notice Exchange zUSD for HZN
     * @param synthAmount The amount of synths the user wishes to exchange.
     */
    function exchangeSynthsForSNX(uint synthAmount)
        external
        rateNotInvalid(HZN)
        notPaused
        returns (
            uint // Returns the number of HZN received
        )
    {
        return _exchangeSynthsForSNX(synthAmount);
    }

    /**
     * @notice Exchange zUSD for HZN while insisting on a particular rate. This allows a user to
     *          exchange while protecting against frontrunning by the contract owner on the exchange rate.
     * @param synthAmount The amount of synths the user wishes to exchange.
     * @param guaranteedRate A rate (synthetix price) the caller wishes to insist upon.
     */
    function exchangeSynthsForSNXAtRate(uint synthAmount, uint guaranteedRate)
        external
        rateNotInvalid(HZN)
        notPaused
        returns (
            uint // Returns the number of HZN received
        )
    {
        require(guaranteedRate == exchangeRates().rateForCurrency(HZN), "Guaranteed rate would not be received");

        return _exchangeSynthsForSNX(synthAmount);
    }

    /**
     * @notice Allows the owner to withdraw HZN from this contract if needed.
     * @param amount The amount of HZN to attempt to withdraw (in 18 decimal places).
     */
    function withdrawSynthetix(uint amount) external onlyOwner {
        synthetix().transfer(owner, amount);

        // We don't emit our own events here because we assume that anyone
        // who wants to watch what the Depot is doing can
        // just watch ERC20 events from the Synth and/or Synthetix contracts
        // filtered to our address.
    }

    /**
     * @notice Allows a user to withdraw all of their previously deposited synths from this contract if needed.
     *          Developer note: We could keep an index of address to deposits to make this operation more efficient
     *          but then all the other operations on the queue become less efficient. It's expected that this
     *          function will be very rarely used, so placing the inefficiency here is intentional. The usual
     *          use case does not involve a withdrawal.
     */
    function withdrawMyDepositedSynths() external {
        uint synthsToSend = 0;

        for (uint i = depositStartIndex; i < depositEndIndex; i++) {
            SynthDepositEntry memory deposit = deposits[i];

            if (deposit.user == msg.sender) {
                // The user is withdrawing this deposit. Remove it from our queue.
                // We'll just leave a gap, which the purchasing logic can walk past.
                synthsToSend = synthsToSend.add(deposit.amount);
                delete deposits[i];
                //Let the DApps know we've removed this deposit
                emit SynthDepositRemoved(deposit.user, deposit.amount, i);
            }
        }

        // Update our total
        totalSellableDeposits = totalSellableDeposits.sub(synthsToSend);
```

```
        // Check if the user has tried to send deposit amounts < the minimumDepositAmount to the FIFO
        // queue which would have been added to this mapping for withdrawal only
        synthsToSend = synthsToSend.add(smallDeposits[msg.sender]);
        smallDeposits[msg.sender] = 0;

        // If there's nothing to do then go ahead and revert the transaction
        require(synthsToSend > 0, "You have no deposits to withdraw.");

        // Send their deposits back to them (minus fees)
        synthsUSD().transfer(msg.sender, synthsToSend);

        emit SynthWithdrawal(msg.sender, synthsToSend);
    }

    /**
     * @notice depositSynths: Allows users to deposit synths via the approve / transferFrom workflow
     * @param amount The amount of zUSD you wish to deposit (must have been approved first)
     */
    function depositSynths(uint amount) external {
        // Grab the amount of synths. Will fail if not approved first
        synthsUSD().transferFrom(msg.sender, address(this), amount);

        // A minimum deposit amount is designed to protect purchasers from over paying
        // gas for fullfilling multiple small synth deposits
        if (amount < minimumDepositAmount) {
            // We cant fail/revert the transaction or send the synths back in a reentrant call.
            // So we will keep your synths balance seperate from the FIFO queue so you can withdraw them
            smallDeposits[msg.sender] = smallDeposits[msg.sender].add(amount);

            emit SynthDepositNotAccepted(msg.sender, amount, minimumDepositAmount);
        } else {
            // Ok, thanks for the deposit, let's queue it up.
            deposits[depositEndIndex] = SynthDepositEntry({user: msg.sender, amount: amount});
            emit SynthDeposit(msg.sender, amount, depositEndIndex);

            // Walk our index forward as well.
            depositEndIndex = depositEndIndex.add(1);

            // And add it to our total.
            totalSellableDeposits = totalSellableDeposits.add(amount);
        }
    }

    /* ========== VIEWS ========== */

    /**
     * @notice Calculate how many HZN you will receive if you transfer
     *          an amount of synths.
     * @param amount The amount of synths (in 18 decimal places) you want to ask about
     */
    function synthetixReceivedForSynths(uint amount) public view returns (uint) {
        // And what would that be worth in HZN based on the current price?
        return amount.divideDecimal(exchangeRates().rateForCurrency(HZN));
    }

    /**
     * @notice Calculate how many HZN you will receive if you transfer
     *          an amount of ether.
     * @param amount The amount of ether (in wei) you want to ask about
     */
    function synthetixReceivedForEther(uint amount) public view returns (uint) {
        // How much is the BNB they sent us worth in zUSD (ignoring the transfer fee)?
        uint valueSentInSynths = amount.multiplyDecimal(exchangeRates().rateForCurrency(BNB));

        // Now, how many HZN will that USD amount buy?
        return synthetixReceivedForSynths(valueSentInSynths);
    }

    /**
     * @notice Calculate how many synths you will receive if you transfer
     *          an amount of ether.
     * @param amount The amount of ether (in wei) you want to ask about
     */
    function synthsReceivedForEther(uint amount) public view returns (uint) {
        // How many synths would that amount of ether be worth?
        return amount.multiplyDecimal(exchangeRates().rateForCurrency(BNB));
    }

    /* ========== INTERNAL VIEWS ========== */

    function synthsUSD() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_ZASSETZUSD, "Missing ZassetzUSD address"));
    }

    function synthetix() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }
```

```
    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    // ========== MODIFIERS ==========

    modifier rateNotInvalid(bytes32 currencyKey) {
        require(!exchangeRates().rateIsInvalid(currencyKey), "Rate invalid or not a zasset");
        _;
    }

    /* ========== EVENTS ========== */

    event MaxEthPurchaseUpdated(uint amount);
    event FundsWalletUpdated(address newFundsWallet);
    event Exchange(string fromCurrency, uint fromAmount, string toCurrency, uint toAmount);
    event SynthWithdrawal(address user, uint amount);
    event SynthDeposit(address indexed user, uint amount, uint indexed depositIndex);
    event SynthDepositRemoved(address indexed user, uint amount, uint indexed depositIndex);
    event SynthDepositNotAccepted(address user, uint amount, uint minimum);
    event MinimumDepositAmountUpdated(uint amount);
    event NonPayableContract(address indexed receiver, uint amount);
    event ClearedDeposit(
        address indexed fromAddress,
        address indexed toAddress,
        uint fromETHAmount,
        uint toAmount,
        uint indexed depositIndex
    );
}
```

EmptyEtherCollateral.sol

```
pragma solidity ^0.5.16;


// Empty contract for ether collateral placeholder for OVM
// https://docs.synthetix.io/contracts/source/contracts/emptyethercollateral
contract EmptyEtherCollateral {
    function totalIssuedSynths() external pure returns (uint) {
        return 0;
    }
}
```

EscrowChecker.sol

```
pragma solidity ^0.5.16;


interface ISynthetixEscrow {
    function numVestingEntries(address account) external view returns (uint);

    function getVestingScheduleEntry(address account, uint index) external view returns (uint[2] memory);
}


// https://docs.synthetix.io/contracts/source/contracts/escrowchecker
contract EscrowChecker {
    ISynthetixEscrow public synthetix_escrow;

    constructor(ISynthetixEscrow _esc) public {
        synthetix_escrow = _esc;
    }

    function checkAccountSchedule(address account) public view returns (uint[16] memory) {
        uint[16] memory _result;
        uint schedules = synthetix_escrow.numVestingEntries(account);
        for (uint i = 0; i < schedules; i++) {
            uint[2] memory pair = synthetix_escrow.getVestingScheduleEntry(account, i);
            _result[i * 2] = pair[0];
            _result[i * 2 + 1] = pair[1];
        }
        return _result;
    }
}
```

EternalStorage.sol

```
pragma solidity ^0.5.16;

// Inheritance
```

```
import "./Owned.sol";
import "./State.sol";


// https://docs.synthetix.io/contracts/source/contracts/eternalstorage
/**
 * @notice    This contract is based on the code available from this blog
 * https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88/
 * Implements support for storing a keccak256 key and value pairs. It is the more flexible
 * and extensible option. This ensures data schema changes can be implemented without
 * requiring upgrades to the storage contract.
 */
contract EternalStorage is Owned, State {
    constructor(address _owner, address _associatedContract) public Owned(_owner) State(_associatedContract)
    {}

    /* ========== DATA TYPES ========== */
    mapping(bytes32 => uint) internal UIntStorage;
    mapping(bytes32 => string) internal StringStorage;
    mapping(bytes32 => address) internal AddressStorage;
    mapping(bytes32 => bytes) internal BytesStorage;
    mapping(bytes32 => bytes32) internal Bytes32Storage;
    mapping(bytes32 => bool) internal BooleanStorage;
    mapping(bytes32 => int) internal IntStorage;

    // UIntStorage;
    function getUIntValue(bytes32 record) external view returns (uint) {
        return UIntStorage[record];
    }

    function setUIntValue(bytes32 record, uint value) external onlyAssociatedContract {
        UIntStorage[record] = value;
    }

    function deleteUIntValue(bytes32 record) external onlyAssociatedContract {
        delete UIntStorage[record];
    }

    // StringStorage
    function getStringValue(bytes32 record) external view returns (string memory) {
        return StringStorage[record];
    }

    function setStringValue(bytes32 record, string calldata value) external onlyAssociatedContract {
        StringStorage[record] = value;
    }

    function deleteStringValue(bytes32 record) external onlyAssociatedContract {
        delete StringStorage[record];
    }

    // AddressStorage
    function getAddressValue(bytes32 record) external view returns (address) {
        return AddressStorage[record];
    }

    function setAddressValue(bytes32 record, address value) external onlyAssociatedContract {
        AddressStorage[record] = value;
    }

    function deleteAddressValue(bytes32 record) external onlyAssociatedContract {
        delete AddressStorage[record];
    }

    // BytesStorage
    function getBytesValue(bytes32 record) external view returns (bytes memory) {
        return BytesStorage[record];
    }

    function setBytesValue(bytes32 record, bytes calldata value) external onlyAssociatedContract {
        BytesStorage[record] = value;
    }

    function deleteBytesValue(bytes32 record) external onlyAssociatedContract {
        delete BytesStorage[record];
    }

    // Bytes32Storage
    function getBytes32Value(bytes32 record) external view returns (bytes32) {
        return Bytes32Storage[record];
    }

    function setBytes32Value(bytes32 record, bytes32 value) external onlyAssociatedContract {
        Bytes32Storage[record] = value;
    }

    function deleteBytes32Value(bytes32 record) external onlyAssociatedContract {
```

```
        delete Bytes32Storage[record];
    }

    // BooleanStorage
    function getBooleanValue(bytes32 record) external view returns (bool) {
        return BooleanStorage[record];
    }

    function setBooleanValue(bytes32 record, bool value) external onlyAssociatedContract {
        BooleanStorage[record] = value;
    }

    function deleteBooleanValue(bytes32 record) external onlyAssociatedContract {
        delete BooleanStorage[record];
    }

    // IntStorage
    function getIntValue(bytes32 record) external view returns (int) {
        return IntStorage[record];
    }

    function setIntValue(bytes32 record, int value) external onlyAssociatedContract {
        IntStorage[record] = value;
    }

    function deleteIntValue(bytes32 record) external onlyAssociatedContract {
        delete IntStorage[record];
    }
}
```

EtherCollateral.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Pausable.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";
import "./MixinResolver.sol";
import "./interfaces/IEtherCollateral.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/ISynth.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/IDepot.sol";
import "./interfaces/IExchangeRates.sol";


// https://docs.synthetix.io/contracts/source/contracts/ethercollateral
contract EtherCollateral is Owned, Pausable, ReentrancyGuard, MixinResolver, IEtherCollateral {
    using SafeMath for uint256;
    using SafeDecimalMath for uint256;

    // ========== CONSTANTS ==========
    uint256 internal constant ONE_THOUSAND = 1e18 * 1000;
    uint256 internal constant ONE_HUNDRED = 1e18 * 100;

    uint256 internal constant SECONDS_IN_A_YEAR = 31536000; // Common Year

    // Where fees are pooled in zUSD.
    address internal constant FEE_ADDRESS = 0xfeEFEEfeefEeFeefEEFEEfEeFeefEEFeeFEEFEeF;

    // ========== SETTER STATE VARIABLES ==========

    // The ratio of Collateral to synths issued
    uint256 public collateralizationRatio = SafeDecimalMath.unit() * 125; // SCCP-27

    // If updated, all outstanding loans will pay this interest rate in on closure of the loan. Default 5%
    uint256 public interestRate = (5 * SafeDecimalMath.unit()) / 100;
    uint256 public interestPerSecond = interestRate.div(SECONDS_IN_A_YEAR);

    // Minting fee for issuing the synths. Default 50 bips.
    uint256 public issueFeeRate = (5 * SafeDecimalMath.unit()) / 1000;

    // Maximum amount of zBNB that can be issued by the EtherCollateral contract. Default 5000
    uint256 public issueLimit = SafeDecimalMath.unit() * 5000;

    // Minimum amount of BNB to create loan preventing griefing and gas consumption. Min 1BNB = 0.8 zBNB
    uint256 public minLoanSize = SafeDecimalMath.unit() * 1;

    // Maximum number of loans an account can create
```

```
    uint256 public accountLoanLimit = 50;

    // If true then any wallet addres can close a loan not just the loan creator.
    bool public loanLiquidationOpen = false;

    // Time when remaining loans can be liquidated
    uint256 public liquidationDeadline;

    // ========== STATE VARIABLES ==========

    // The total number of synths issued by the collateral in this contract
    uint256 public totalIssuedSynths;

    // Total number of loans ever created
    uint256 public totalLoansCreated;

    // Total number of open loans
    uint256 public totalOpenLoanCount;

    // Synth loan storage struct
    struct SynthLoanStruct {
        //  Acccount that created the loan
        address account;
        //  Amount (in collateral token ) that they deposited
        uint256 collateralAmount;
        //  Amount (in synths) that they issued to borrow
        uint256 loanAmount;
        // When the loan was created
        uint256 timeCreated;
        // ID for the loan
        uint256 loanID;
        // When the loan was paidback (closed)
        uint256 timeClosed;
    }

    // Users Loans by address
    mapping(address => SynthLoanStruct[]) public accountsSynthLoans;

    // Account Open Loan Counter
    mapping(address => uint256) public accountOpenLoanCounter;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_ZASSETZBNB = "ZassetzBNB";
    bytes32 private constant CONTRACT_ZASSETZUSD = "ZassetzUSD";
    bytes32 private constant CONTRACT_DEPOT = "Depot";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_ZASSETZBNB,
        CONTRACT_ZASSETZUSD,
        CONTRACT_DEPOT,
        CONTRACT_EXRATES
    ];

    // ========== CONSTRUCTOR ==========
    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        Pausable()
        MixinResolver(_resolver, addressesToCache)
    {
        liquidationDeadline = now + 92 days; // Time before loans can be liquidated
    }

    // ========== SETTERS ==========

    function setCollateralizationRatio(uint256 ratio) external onlyOwner {
        require(ratio <= ONE_THOUSAND, "Too high");
        require(ratio >= ONE_HUNDRED, "Too low");
        collateralizationRatio = ratio;
        emit CollateralizationRatioUpdated(ratio);
    }

    function setInterestRate(uint256 _interestRate) external onlyOwner {
        require(_interestRate > SECONDS_IN_A_YEAR, "Interest rate cannot be less that the
SECONDS_IN_A_YEAR");
        require(_interestRate <= SafeDecimalMath.unit(), "Interest cannot be more than 100% APR");
        interestRate = _interestRate;
        interestPerSecond = _interestRate.div(SECONDS_IN_A_YEAR);
        emit InterestRateUpdated(interestRate);
    }

    function setIssueFeeRate(uint256 _issueFeeRate) external onlyOwner {
        issueFeeRate = _issueFeeRate;
```

```
        emit IssueFeeRateUpdated(issueFeeRate);
    }

    function setIssueLimit(uint256 _issueLimit) external onlyOwner {
        issueLimit = _issueLimit;
        emit IssueLimitUpdated(issueLimit);
    }

    function setMinLoanSize(uint256 _minLoanSize) external onlyOwner {
        minLoanSize = _minLoanSize;
        emit MinLoanSizeUpdated(minLoanSize);
    }

    function setAccountLoanLimit(uint256 _loanLimit) external onlyOwner {
        uint256 HARD_CAP = 1000;
        require(_loanLimit < HARD_CAP, "Owner cannot set higher than HARD_CAP");
        accountLoanLimit = _loanLimit;
        emit AccountLoanLimitUpdated(accountLoanLimit);
    }

    function setLoanLiquidationOpen(bool _loanLiquidationOpen) external onlyOwner {
        require(now > liquidationDeadline, "Before liquidation deadline");
        loanLiquidationOpen = _loanLiquidationOpen;
        emit LoanLiquidationOpenUpdated(loanLiquidationOpen);
    }

    // ========== PUBLIC VIEWS ==========

    function getContractInfo()
        external
        view
        returns (
            uint256 _collateralizationRatio,
            uint256 _issuanceRatio,
            uint256 _interestRate,
            uint256 _interestPerSecond,
            uint256 _issueFeeRate,
            uint256 _issueLimit,
            uint256 _minLoanSize,
            uint256 _totalIssuedSynths,
            uint256 _totalLoansCreated,
            uint256 _totalOpenLoanCount,
            uint256 _ethBalance,
            uint256 _liquidationDeadline,
            bool _loanLiquidationOpen
        )
    {
        _collateralizationRatio = collateralizationRatio;
        _issuanceRatio = issuanceRatio();
        _interestRate = interestRate;
        _interestPerSecond = interestPerSecond;
        _issueFeeRate = issueFeeRate;
        _issueLimit = issueLimit;
        _minLoanSize = minLoanSize;
        _totalIssuedSynths = totalIssuedSynths;
        _totalLoansCreated = totalLoansCreated;
        _totalOpenLoanCount = totalOpenLoanCount;
        _ethBalance = address(this).balance;
        _liquidationDeadline = liquidationDeadline;
        _loanLiquidationOpen = loanLiquidationOpen;
    }

    // returns value of 100 / collateralizationRatio.
    // e.g. 100/125 = 0.8
    // or in wei 100000000000000000000/125000000000000000000 = 800000000000000000
    function issuanceRatio() public view returns (uint256) {
        // this Rounds so you get slightly more rather than slightly less
        // 4999999999999999995000
        return ONE_HUNDRED.divideDecimalRound(collateralizationRatio);
    }

    function loanAmountFromCollateral(uint256 collateralAmount) public view returns (uint256) {
        return collateralAmount.multiplyDecimal(issuanceRatio());
    }

    function collateralAmountForLoan(uint256 loanAmount) external view returns (uint256) {
        return loanAmount.multiplyDecimal(collateralizationRatio.divideDecimalRound(ONE_HUNDRED));
    }

    function currentInterestOnLoan(address _account, uint256 _loanID) external view returns (uint256) {
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        uint256 loanLifeSpan = _loanLifeSpan(synthLoan);
        return accruedInterestOnLoan(synthLoan.loanAmount, loanLifeSpan);
    }

    function accruedInterestOnLoan(uint256 _loanAmount, uint256 _seconds) public view returns (uint256
```

```
interestAmount) {
        // Simple interest calculated per second
        // Interest = Principal * rate * time
        interestAmount = _loanAmount.multiplyDecimalRound(interestPerSecond.mul(_seconds));
    }

    function calculateMintingFee(address _account, uint256 _loanID) external view returns (uint256) {
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        return _calculateMintingFee(synthLoan);
    }

    function openLoanIDsByAccount(address _account) external view returns (uint256[] memory) {
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[_account];

        uint256[] memory _openLoanIDs = new uint256[](synthLoans.length);
        uint256 _counter = 0;

        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].timeClosed == 0) {
                _openLoanIDs[_counter] = synthLoans[i].loanID;
                _counter++;
            }
        }
        // Create the fixed size array to return
        uint256[] memory _result = new uint256[](_counter);

        // Copy loanIDs from dynamic array to fixed array
        for (uint256 j = 0; j < _counter; j++) {
            _result[j] = _openLoanIDs[j];
        }
        // Return an array with list of open Loan IDs
        return _result;
    }

    function getLoan(address _account, uint256 _loanID)
        external
        view
        returns (
            address account,
            uint256 collateralAmount,
            uint256 loanAmount,
            uint256 timeCreated,
            uint256 loanID,
            uint256 timeClosed,
            uint256 interest,
            uint256 totalFees
        )
    {
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        account = synthLoan.account;
        collateralAmount = synthLoan.collateralAmount;
        loanAmount = synthLoan.loanAmount;
        timeCreated = synthLoan.timeCreated;
        loanID = synthLoan.loanID;
        timeClosed = synthLoan.timeClosed;
        interest = accruedInterestOnLoan(synthLoan.loanAmount, _loanLifeSpan(synthLoan));
        totalFees = interest.add(_calculateMintingFee(synthLoan));
    }

    function loanLifeSpan(address _account, uint256 _loanID) external view returns (uint256 loanLifeSpanResult)
{
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        loanLifeSpanResult = _loanLifeSpan(synthLoan);
    }

    // ========== PUBLIC FUNCTIONS ==========

    function openLoan() external payable notPaused nonReentrant sETHRateNotInvalid returns (uint256 loanID)
{
        systemStatus().requireIssuanceActive();

        // Require ETH sent to be greater than minLoanSize
        require(msg.value >= minLoanSize, "Not enough BNB to create this loan. Please see the minLoanSize");

        // Require loanLiquidationOpen to be false or we are in liquidation phase
        require(loanLiquidationOpen == false, "Loans are now being liquidated");

        // Each account is limted to creating 50 (accountLoanLimit) loans
        require(accountsSynthLoans[msg.sender].length < accountLoanLimit, "Each account is limted to 50
loans");

        // Calculate issuance amount
        uint256 loanAmount = loanAmountFromCollateral(msg.value);

        // Require zBNB to mint does not exceed cap
        require(totalIssuedSynths.add(loanAmount) < issueLimit, "Loan Amount exceeds the supply cap.");
```

```
        // Get a Loan ID
        loanID = _incrementTotalLoansCounter();

        // Create Loan storage object
        SynthLoanStruct memory synthLoan = SynthLoanStruct({
            account: msg.sender,
            collateralAmount: msg.value,
            loanAmount: loanAmount,
            timeCreated: now,
            loanID: loanID,
            timeClosed: 0
        });

        // Record loan in mapping to account in an array of the accounts open loans
        accountsSynthLoans[msg.sender].push(synthLoan);

        // Increment totalIssuedSynths
        totalIssuedSynths = totalIssuedSynths.add(loanAmount);

        // Issue the synth
        synthsETH().issue(msg.sender, loanAmount);

        // Tell the Dapps a loan was created
        emit LoanCreated(msg.sender, loanID, loanAmount);
    }

    function closeLoan(uint256 loanID) external nonReentrant sETHRateNotInvalid {
        _closeLoan(msg.sender, loanID);
    }

    // Liquidation of an open loan available for anyone
    function liquidateUnclosedLoan(address _loanCreatorsAddress, uint256 _loanID) external nonReentrant
sETHRateNotInvalid {
        require(loanLiquidationOpen, "Liquidation is not open");
        // Close the creators loan and send collateral to the closer.
        _closeLoan(_loanCreatorsAddress, _loanID);
        // Tell the Dapps this loan was liquidated
        emit LoanLiquidated(_loanCreatorsAddress, _loanID, msg.sender);
    }

    // ========== PRIVATE FUNCTIONS ==========

    function _closeLoan(address account, uint256 loanID) private {
        systemStatus().requireIssuanceActive();

        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(account, loanID);

        require(synthLoan.loanID > 0, "Loan does not exist");
        require(synthLoan.timeClosed == 0, "Loan already closed");
        require(
            IERC20(address(synthsETH())).balanceOf(msg.sender) >= synthLoan.loanAmount,
            "You do not have the required Zasset balance to close this loan."
        );

        // Record loan as closed
        _recordLoanClosure(synthLoan);

        // Decrement totalIssuedSynths
        totalIssuedSynths = totalIssuedSynths.sub(synthLoan.loanAmount);

        // Calculate and deduct interest(5%) and minting fee(50 bips) in ETH
        uint256 interestAmount = accruedInterestOnLoan(synthLoan.loanAmount, _loanLifeSpan(synthLoan));
        uint256 mintingFee = _calculateMintingFee(synthLoan);
        uint256 totalFeeETH = interestAmount.add(mintingFee);

        // Burn all Synths issued for the loan
        synthsETH().burn(msg.sender, synthLoan.loanAmount);

        // Fee Distribution. Purchase zUSD with BNB from Depot
        require(
            IERC20(address(synthsUSD())).balanceOf(address(depot()))                                    >=
depot().synthsReceivedForEther(totalFeeETH),
            "The zUSD Depot does not have enough zUSD to buy for fees"
        );
        depot().exchangeEtherForSynths.value(totalFeeETH)();

        // Transfer the zUSD to distribute to HZN holders.
        IERC20(address(synthsUSD())).transfer(FEE_ADDRESS,
IERC20(address(synthsUSD())).balanceOf(address(this)));

        // Send remainder BNB to caller
        address(msg.sender).transfer(synthLoan.collateralAmount.sub(totalFeeETH));

        // Tell the Dapps
        emit LoanClosed(account, loanID, totalFeeETH);
```

```
    }

    function _getLoanFromStorage(address account, uint256 loanID) private view returns (SynthLoanStruct memory) {
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == loanID) {
                return synthLoans[i];
            }
        }
    }

    function _recordLoanClosure(SynthLoanStruct memory synthLoan) private {
        // Get storage pointer to the accounts array of loans
        SynthLoanStruct[] storage synthLoans = accountsSynthLoans[synthLoan.account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == synthLoan.loanID) {
                // Record the time the loan was closed
                synthLoans[i].timeClosed = now;
            }
        }

        // Reduce Total Open Loans Count
        totalOpenLoanCount = totalOpenLoanCount.sub(1);
    }

    function _incrementTotalLoansCounter() private returns (uint256) {
        // Increase the total Open loan count
        totalOpenLoanCount = totalOpenLoanCount.add(1);
        // Increase the total Loans Created count
        totalLoansCreated = totalLoansCreated.add(1);
        // Return total count to be used as a unique ID.
        return totalLoansCreated;
    }

    function _calculateMintingFee(SynthLoanStruct memory synthLoan) private view returns (uint256 mintingFee) {
        mintingFee = synthLoan.loanAmount.multiplyDecimalRound(issueFeeRate);
    }

    function loanLifeSpan(SynthLoanStruct memory synthLoan) private view returns (uint256 loanLifeSpanResult) {
        // Get time loan is open for, and if closed from the timeClosed
        bool loanClosed = synthLoan.timeClosed > 0;
        // Calculate loan life span in seconds as (Now - Loan creation time)
        loanLifeSpanResult = loanClosed ? synthLoan.timeClosed.sub(synthLoan.timeCreated) : now.sub(synthLoan.timeCreated);
    }

    /* ========== INTERNAL VIEWS ========== */

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function synthsETH() internal view returns (ISynth) {
        return ISynth(requireAndGetAddress(CONTRACT_ZASSETZBNB, "Missing ZassetzBNB address"));
    }

    function synthsUSD() internal view returns (ISynth) {
        return ISynth(requireAndGetAddress(CONTRACT_ZASSETZUSD, "Missing ZassetzUSD address"));
    }

    function depot() internal view returns (IDepot) {
        return IDepot(requireAndGetAddress(CONTRACT_DEPOT, "Missing Depot address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    /* ========== MODIFIERS ========== */

    modifier sETHRateNotInvalid() {
        require(!exchangeRates().rateIsInvalid("zBNB"), "Blocked as zBNB rate is invalid");
        _;
    }

    // ========== EVENTS ==========

    event CollateralizationRatioUpdated(uint256 ratio);
    event InterestRateUpdated(uint256 interestRate);
    event IssueFeeRateUpdated(uint256 issueFeeRate);
    event IssueLimitUpdated(uint256 issueLimit);
    event MinLoanSizeUpdated(uint256 minLoanSize);
```

```
        event AccountLoanLimitUpdated(uint256 loanLimit);
        event LoanLiquidationOpenUpdated(bool loanLiquidationOpen);
        event LoanCreated(address indexed account, uint256 loanID, uint256 amount);
        event LoanClosed(address indexed account, uint256 loanID, uint256 feesPaid);
        event LoanLiquidated(address indexed account, uint256 loanID, address liquidator);
}
```

EtherCollateralsUSD.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Pausable.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";
import "./MixinResolver.sol";
import "./interfaces/IEtherCollateralsUSD.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/ISynth.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/IExchangeRates.sol";


// ETH Collateral v0.3 (zUSD)
// https://docs.synthetix.io/contracts/source/contracts/ethercollateralsusd
contract EtherCollateralsUSD is Owned, Pausable, ReentrancyGuard, MixinResolver, IEtherCollateralsUSD {
        using SafeMath for uint256;
        using SafeDecimalMath for uint256;

        bytes32 internal constant BNB = "BNB";

        // ========== CONSTANTS ==========
        uint256 internal constant ONE_THOUSAND = 1e18 * 1000;
        uint256 internal constant ONE_HUNDRED = 1e18 * 100;

        uint256 internal constant SECONDS_IN_A_YEAR = 31536000; // Common Year

        // Where fees are pooled in zUSD.
        address internal constant FEE_ADDRESS = 0xfeEFEEfeefEeFeefEEFEEfEeFeefEEFeeFEEFEeF;

        uint256 internal constant ACCOUNT_LOAN_LIMIT_CAP = 1000;
        bytes32 private constant zUSD = "zUSD";
        bytes32 public constant COLLATERAL = "BNB";

        // ========== SETTER STATE VARIABLES ==========

        // The ratio of Collateral to synths issued
        uint256 public collateralizationRatio = SafeDecimalMath.unit() * 150;

        // If updated, all outstanding loans will pay this interest rate in on closure of the loan. Default 5%
        uint256 public interestRate = (5 * SafeDecimalMath.unit()) / 100;
        uint256 public interestPerSecond = interestRate.div(SECONDS_IN_A_YEAR);

        // Minting fee for issuing the synths. Default 50 bips.
        uint256 public issueFeeRate = (5 * SafeDecimalMath.unit()) / 1000;

        // Maximum amount of zUSD that can be issued by the EtherCollateral contract. Default 10MM
        uint256 public issueLimit = SafeDecimalMath.unit() * 10000000;

        // Minimum amount of BNB to create loan preventing griefing and gas consumption. Min 1BNB
        uint256 public minLoanCollateralSize = SafeDecimalMath.unit() * 1;

        // Maximum number of loans an account can create
        uint256 public accountLoanLimit = 50;

        // If true then any wallet addres can close a loan not just the loan creator.
        bool public loanLiquidationOpen = false;

        // Time when remaining loans can be liquidated
        uint256 public liquidationDeadline;

        // Liquidation ratio when loans can be liquidated
        uint256 public liquidationRatio = (150 * SafeDecimalMath.unit()) / 100; // 1.5 ratio

        // Liquidation penalty when loans are liquidated. default 10%
        uint256 public liquidationPenalty = SafeDecimalMath.unit() / 10;

        // ========== STATE VARIABLES ==========
```

```
    // The total number of synths issued by the collateral in this contract
    uint256 public totalIssuedSynths;

    // Total number of loans ever created
    uint256 public totalLoansCreated;

    // Total number of open loans
    uint256 public totalOpenLoanCount;

    // Synth loan storage struct
    struct SynthLoanStruct {
        //  Acccount that created the loan
        address payable account;
        //  Amount (in collateral token ) that they deposited
        uint256 collateralAmount;
        //  Amount (in synths) that they issued to borrow
        uint256 loanAmount;
        // Minting Fee
        uint256 mintingFee;
        // When the loan was created
        uint256 timeCreated;
        // ID for the loan
        uint256 loanID;
        // When the loan was paidback (closed)
        uint256 timeClosed;
        // Applicable Interest rate
        uint256 loanInterestRate;
        // interest amounts accrued
        uint256 accruedInterest;
        // last timestamp interest amounts accrued
        uint40 lastInterestAccrued;
    }

    // Users Loans by address
    mapping(address => SynthLoanStruct[]) public accountsSynthLoans;

    // Account Open Loan Counter
    mapping(address => uint256) public accountOpenLoanCounter;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_ZASSETZUSD = "ZassetzUSD";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 private constant CONTRACT_FEEPOOL = "FeePool";

    bytes32[24] private addressesToCache = [CONTRACT_SYSTEMSTATUS, CONTRACT_ZASSETZUSD,
CONTRACT_EXRATES, CONTRACT_FEEPOOL];

    // ========== CONSTRUCTOR ==========
    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        Pausable()
        MixinResolver(_resolver, addressesToCache)
    {
        liquidationDeadline = block.timestamp + 92 days; // Time before loans can be open for liquidation to end
the trial contract
    }
    // ========== SETTERS ==========

    function setCollateralizationRatio(uint256 ratio) external onlyOwner {
        require(ratio <= ONE_THOUSAND, "Too high");
        require(ratio >= ONE_HUNDRED, "Too low");
        collateralizationRatio = ratio;
        emit CollateralizationRatioUpdated(ratio);
    }

    function setInterestRate(uint256 _interestRate) external onlyOwner {
        require(_interestRate > SECONDS_IN_A_YEAR, "Interest rate cannot be less that the
SECONDS_IN_A_YEAR");
        require(_interestRate <= SafeDecimalMath.unit(), "Interest cannot be more than 100% APR");
        interestRate = _interestRate;
        interestPerSecond = _interestRate.div(SECONDS_IN_A_YEAR);
        emit InterestRateUpdated(interestRate);
    }

    function setIssueFeeRate(uint256 _issueFeeRate) external onlyOwner {
        issueFeeRate = _issueFeeRate;
        emit IssueFeeRateUpdated(issueFeeRate);
    }

    function setIssueLimit(uint256 _issueLimit) external onlyOwner {
        issueLimit = _issueLimit;
        emit IssueLimitUpdated(issueLimit);
    }
```

```
    function setMinLoanCollateralSize(uint256 _minLoanCollateralSize) external onlyOwner {
        minLoanCollateralSize = _minLoanCollateralSize;
        emit MinLoanCollateralSizeUpdated(minLoanCollateralSize);
    }

    function setAccountLoanLimit(uint256 _loanLimit) external onlyOwner {
        require(_loanLimit < ACCOUNT_LOAN_LIMIT_CAP, "Owner cannot set higher than
ACCOUNT_LOAN_LIMIT_CAP");
        accountLoanLimit = _loanLimit;
        emit AccountLoanLimitUpdated(accountLoanLimit);
    }

    function setLoanLiquidationOpen(bool _loanLiquidationOpen) external onlyOwner {
        require(block.timestamp > liquidationDeadline, "Before liquidation deadline");
        loanLiquidationOpen = _loanLiquidationOpen;
        emit LoanLiquidationOpenUpdated(loanLiquidationOpen);
    }

    function setLiquidationRatio(uint256 _liquidationRatio) external onlyOwner {
        require(_liquidationRatio > SafeDecimalMath.unit(), "Ratio less than 100%");
        liquidationRatio = _liquidationRatio;
        emit LiquidationRatioUpdated(liquidationRatio);
    }

    // ========== PUBLIC VIEWS ==========

    function getContractInfo()
        external
        view
        returns (
            uint256 _collateralizationRatio,
            uint256 _issuanceRatio,
            uint256 _interestRate,
            uint256 _interestPerSecond,
            uint256 _issueFeeRate,
            uint256 _issueLimit,
            uint256 _minLoanCollateralSize,
            uint256 _totalIssuedSynths,
            uint256 _totalLoansCreated,
            uint256 _totalOpenLoanCount,
            uint256 _ethBalance,
            uint256 _liquidationDeadline,
            bool _loanLiquidationOpen
        )
    {
        _collateralizationRatio = collateralizationRatio;
        _issuanceRatio = issuanceRatio();
        _interestRate = interestRate;
        _interestPerSecond = interestPerSecond;
        _issueFeeRate = issueFeeRate;
        _issueLimit = issueLimit;
        _minLoanCollateralSize = minLoanCollateralSize;
        _totalIssuedSynths = totalIssuedSynths;
        _totalLoansCreated = totalLoansCreated;
        _totalOpenLoanCount = totalOpenLoanCount;
        _ethBalance = address(this).balance;
        _liquidationDeadline = liquidationDeadline;
        _loanLiquidationOpen = loanLiquidationOpen;
    }

    // returns value of 100 / collateralizationRatio.
    // e.g. 100/150 = 0.6666666667
    function issuanceRatio() public view returns (uint256) {
        // this rounds so you get slightly more rather than slightly less
        return ONE_HUNDRED.divideDecimalRound(collateralizationRatio);
    }

    function loanAmountFromCollateral(uint256 collateralAmount) public view returns (uint256) {
        // a fraction more is issued due to rounding
        return
collateralAmount.multiplyDecimal(issuanceRatio()).multiplyDecimal(exchangeRates().rateForCurrency(BNB));
    }

    function collateralAmountForLoan(uint256 loanAmount) external view returns (uint256) {
        return
            loanAmount
                .multiplyDecimal(collateralizationRatio.divideDecimalRound(exchangeRates().rateForCurren
cy(BNB)))
                .divideDecimalRound(ONE_HUNDRED);
    }

    // compound accrued interest with remaining loanAmount * (now - lastTimestampInterestPaid)
    function currentInterestOnLoan(address _account, uint256 _loanID) external view returns (uint256) {
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        uint256 currentInterest = accruedInterestOnLoan(
```

```
            synthLoan.loanAmount.add(synthLoan.accruedInterest),
            _timeSinceInterestAccrual(synthLoan)
        );
        return synthLoan.accruedInterest.add(currentInterest);
    }

    function accruedInterestOnLoan(uint256 _loanAmount, uint256 _seconds) public view returns (uint256 interestAmount) {
        // Simple interest calculated per second
        // Interest = Principal * rate * time
        interestAmount = _loanAmount.multiplyDecimalRound(interestPerSecond.mul(_seconds));
    }

    function totalFeesOnLoan(address _account, uint256 _loanID)
        external
        view
        returns (uint256 interestAmount, uint256 mintingFee)
    {
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        uint256 loanAmountWithAccruedInterest = synthLoan.loanAmount.add(synthLoan.accruedInterest);
        interestAmount = synthLoan.accruedInterest.add(
            accruedInterestOnLoan(loanAmountWithAccruedInterest, _timeSinceInterestAccrual(synthLoan))
        );
        mintingFee = synthLoan.mintingFee;
    }

    function getMintingFee(address _account, uint256 _loanID) external view returns (uint256) {
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
        return synthLoan.mintingFee;
    }

    /**
     * r = target issuance ratio
     * D = debt balance
     * V = Collateral
     * P = liquidation penalty
     * Calculates amount of synths = (D - V * r) / (1 - (1 + P) * r)
     */
    function calculateAmountToLiquidate(uint debtBalance, uint collateral) public view returns (uint) {
        uint unit = SafeDecimalMath.unit();
        uint ratio = liquidationRatio;

        uint dividend = debtBalance.sub(collateral.divideDecimal(ratio));
        uint divisor = unit.sub(unit.add(liquidationPenalty).divideDecimal(ratio));

        return dividend.divideDecimal(divisor);
    }

    function openLoanIDsByAccount(address _account) external view returns (uint256[] memory) {
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[_account];

        uint256[] memory _openLoanIDs = new uint256[](synthLoans.length);
        uint256 _counter = 0;

        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].timeClosed == 0) {
                _openLoanIDs[_counter] = synthLoans[i].loanID;
                _counter++;
            }
        }
        // Create the fixed size array to return
        uint256[] memory _result = new uint256[](_counter);

        // Copy loanIDs from dynamic array to fixed array
        for (uint256 j = 0; j < _counter; j++) {
            _result[j] = _openLoanIDs[j];
        }
        // Return an array with list of open Loan IDs
        return _result;
    }

    function getLoan(address _account, uint256 _loanID)
        external
        view
        returns (
            address account,
            uint256 collateralAmount,
            uint256 loanAmount,
            uint256 timeCreated,
            uint256 loanID,
            uint256 timeClosed,
            uint256 accruedInterest,
            uint256 totalFees
        )
    {
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);
```

```
        account = synthLoan.account;
        collateralAmount = synthLoan.collateralAmount;
        loanAmount = synthLoan.loanAmount;
        timeCreated = synthLoan.timeCreated;
        loanID = synthLoan.loanID;
        timeClosed = synthLoan.timeClosed;
        accruedInterest = synthLoan.accruedInterest.add(
            accruedInterestOnLoan(synthLoan.loanAmount.add(synthLoan.accruedInterest),
_timeSinceInterestAccrual(synthLoan))
        );
        totalFees = accruedInterest.add(synthLoan.mintingFee);
    }

    function getLoanCollateralRatio(address _account, uint256 _loanID) external view returns (uint256
loanCollateralRatio) {
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);

        (loanCollateralRatio, , ) = _loanCollateralRatio(synthLoan);
    }

    function _loanCollateralRatio(SynthLoanStruct memory _loan)
        internal
        view
        returns (
            uint256 loanCollateralRatio,
            uint256 collateralValue,
            uint256 interestAmount
        )
    {
        // Any interest accrued prior is rolled up into loan amount
        uint256 loanAmountWithAccruedInterest = _loan.loanAmount.add(_loan.accruedInterest);

        interestAmount                =                accruedInterestOnLoan(loanAmountWithAccruedInterest,
_timeSinceInterestAccrual(_loan));

        collateralValue                                                                                =
_loan.collateralAmount.multiplyDecimal(exchangeRates().rateForCurrency(COLLATERAL));

        loanCollateralRatio                                                                            =
collateralValue.divideDecimal(loanAmountWithAccruedInterest.add(interestAmount));
    }

    function timeSinceInterestAccrualOnLoan(address _account, uint256 _loanID) external view returns (uint256)
{
        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_account, _loanID);

        return _timeSinceInterestAccrual(synthLoan);
    }

    // ========== PUBLIC FUNCTIONS ==========

    function openLoan(uint256 _loanAmount)
        external
        payable
        notPaused
        nonReentrant
        ETHRateNotInvalid
        returns (uint256 loanID)
    {
        systemStatus().requireIssuanceActive();

        // Require BNB sent to be greater than minLoanCollateralSize
        require(
            msg.value >= minLoanCollateralSize,
            "Not enough BNB to create this loan. Please see the minLoanCollateralSize"
        );

        // Require loanLiquidationOpen to be false or we are in liquidation phase
        require(loanLiquidationOpen == false, "Loans are now being liquidated");

        // Each account is limited to creating 50 (accountLoanLimit) loans
        require(accountsSynthLoans[msg.sender].length < accountLoanLimit, "Each account is limited to 50
loans");

        // Calculate issuance amount based on issuance ratio
        uint256 maxLoanAmount = loanAmountFromCollateral(msg.value);

        // Require requested _loanAmount to be less than maxLoanAmount
        // Issuance ratio caps collateral to loan value at 150%
        require(_loanAmount <= maxLoanAmount, "Loan amount exceeds max borrowing power");

        uint256 mintingFee = _calculateMintingFee(_loanAmount);
        uint256 loanAmountMinusFee = _loanAmount.sub(mintingFee);

        // Require zUSD loan to mint does not exceed cap
```

```
        require(totalIssuedSynths.add(_loanAmount) <= issueLimit, "Loan Amount exceeds the supply cap.");

        // Get a Loan ID
        loanID = _incrementTotalLoansCounter();

        // Create Loan storage object
        SynthLoanStruct memory synthLoan = SynthLoanStruct({
            account: msg.sender,
            collateralAmount: msg.value,
            loanAmount: _loanAmount,
            mintingFee: mintingFee,
            timeCreated: block.timestamp,
            loanID: loanID,
            timeClosed: 0,
            loanInterestRate: interestRate,
            accruedInterest: 0,
            lastInterestAccrued: 0
        });

        // Fee distribution. Mint the zUSD fees into the FeePool and record fees paid
        if (mintingFee > 0) {
            synthsUSD().issue(FEE_ADDRESS, mintingFee);
            feePool().recordFeePaid(mintingFee);
        }

        // Record loan in mapping to account in an array of the accounts open loans
        accountsSynthLoans[msg.sender].push(synthLoan);

        // Increment totalIssuedSynths
        totalIssuedSynths = totalIssuedSynths.add(_loanAmount);

        // Issue the synth (less fee)
        synthsUSD().issue(msg.sender, loanAmountMinusFee);

        // Tell the Dapps a loan was created
        emit LoanCreated(msg.sender, loanID, _loanAmount);
    }

    function closeLoan(uint256 loanID) external nonReentrant ETHRateNotInvalid {
        _closeLoan(msg.sender, loanID, false);
    }

    // Add BNB collateral to an open loan
    function depositCollateral(address account, uint256 loanID) external payable notPaused {
        require(msg.value > 0, "Deposit amount must be greater than 0");

        systemStatus().requireIssuanceActive();

        // Require loanLiquidationOpen to be false or we are in liquidation phase
        require(loanLiquidationOpen == false, "Loans are now being liquidated");

        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(account, loanID);

        // Check loan exists and is open
        _checkLoanIsOpen(synthLoan);

        uint256 totalCollateral = synthLoan.collateralAmount.add(msg.value);

        _updateLoanCollateral(synthLoan, totalCollateral);

        // Tell the Dapps collateral was added to loan
        emit CollateralDeposited(account, loanID, msg.value, totalCollateral);
    }

    // Withdraw BNB collateral from an open loan
    function withdrawCollateral(uint256 loanID, uint256 withdrawAmount) external notPaused nonReentrant
ETHRateNotInvalid {
        require(withdrawAmount > 0, "Amount to withdraw must be greater than 0");

        systemStatus().requireIssuanceActive();

        // Require loanLiquidationOpen to be false or we are in liquidation phase
        require(loanLiquidationOpen == false, "Loans are now being liquidated");

        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(msg.sender, loanID);

        // Check loan exists and is open
        _checkLoanIsOpen(synthLoan);

        uint256 collateralAfter = synthLoan.collateralAmount.sub(withdrawAmount);

        SynthLoanStruct memory loanAfter = _updateLoanCollateral(synthLoan, collateralAfter);

        // require collateral ratio after to be above the liquidation ratio
        (uint256 collateralRatioAfter, , ) = _loanCollateralRatio(loanAfter);
```

```
        require(collateralRatioAfter > liquidationRatio, "Collateral ratio below liquidation after withdraw");

        // transfer BNB to msg.sender
        msg.sender.transfer(withdrawAmount);

        // Tell the Dapps collateral was added to loan
        emit CollateralWithdrawn(msg.sender, loanID, withdrawAmount, loanAfter.collateralAmount);
    }

    function repayLoan(
        address _loanCreatorsAddress,
        uint256 _loanID,
        uint256 _repayAmount
    ) external ETHRateNotInvalid {
        systemStatus().requireSystemActive();

        // check msg.sender has sufficient zUSD to pay
        require(IERC20(address(synthsUSD())).balanceOf(msg.sender) >= _repayAmount, "Not enough zUSD balance");

        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_loanCreatorsAddress, _loanID);

        // Check loan exists and is open
        _checkLoanIsOpen(synthLoan);

        // Any interest accrued prior is rolled up into loan amount
        uint256 loanAmountWithAccruedInterest = synthLoan.loanAmount.add(synthLoan.accruedInterest);
        uint256 interestAmount = accruedInterestOnLoan(loanAmountWithAccruedInterest, _timeSinceInterestAccrual(synthLoan));

        // repay any accrued interests first
        // and repay principal loan amount with remaining amounts
        uint256 accruedInterest = synthLoan.accruedInterest.add(interestAmount);

        (
            uint256 interestPaid,
            uint256 loanAmountPaid,
            uint256 accruedInterestAfter,
            uint256 loanAmountAfter
        ) = _splitInterestLoanPayment(_repayAmount, accruedInterest, synthLoan.loanAmount);

        // burn zUSD from msg.sender for repaid amount
        synthsUSD().burn(msg.sender, _repayAmount);

        // Send interest paid to fee pool and record loan amount paid
        _processInterestAndLoanPayment(interestPaid, loanAmountPaid);

        // update loan with new total loan amount, record accrued interests
        _updateLoan(synthLoan, loanAmountAfter, accruedInterestAfter, block.timestamp);

        emit LoanRepaid(_loanCreatorsAddress, _loanID, _repayAmount, loanAmountAfter);
    }

    // Liquidate loans at or below issuance ratio
    function liquidateLoan(
        address _loanCreatorsAddress,
        uint256 _loanID,
        uint256 _debtToCover
    ) external nonReentrant ETHRateNotInvalid {
        systemStatus().requireSystemActive();

        // check msg.sender (liquidator's wallet) has sufficient zUSD
        require(IERC20(address(synthsUSD())).balanceOf(msg.sender) >= _debtToCover, "Not enough zUSD balance");

        SynthLoanStruct memory synthLoan = _getLoanFromStorage(_loanCreatorsAddress, _loanID);

        // Check loan exists and is open
        _checkLoanIsOpen(synthLoan);

        (uint256 collateralRatio, uint256 collateralValue, uint256 interestAmount) = _loanCollateralRatio(synthLoan);

        require(collateralRatio < liquidationRatio, "Collateral ratio above liquidation ratio");

        // calculate amount to liquidate to fix ratio including accrued interest
        uint256 liquidationAmount = calculateAmountToLiquidate(
            synthLoan.loanAmount.add(synthLoan.accruedInterest).add(interestAmount),
            collateralValue
        );

        // cap debt to liquidate
        uint256 amountToLiquidate = liquidationAmount < _debtToCover ? liquidationAmount : _debtToCover;

        // burn zUSD from msg.sender for amount to liquidate
        synthsUSD().burn(msg.sender, amountToLiquidate);
```

```
        (uint256 interestPaid, uint256 loanAmountPaid, uint256 accruedInterestAfter, ) =
_splitInterestLoanPayment(
            amountToLiquidate,
            synthLoan.accruedInterest.add(interestAmount),
            synthLoan.loanAmount
        );

        // Send interests paid to fee pool and record loan amount paid
        _processInterestAndLoanPayment(interestPaid, loanAmountPaid);

        // Collateral value to redeem
        uint256 collateralRedeemed = exchangeRates().effectiveValue(zUSD, amountToLiquidate,
COLLATERAL);

        // Add penalty
        uint256 totalCollateralLiquidated = collateralRedeemed.multiplyDecimal(
            SafeDecimalMath.unit().add(liquidationPenalty)
        );

        // update remaining loanAmount less amount paid and update accrued interests less interest paid
        _updateLoan(synthLoan, synthLoan.loanAmount.sub(loanAmountPaid), accruedInterestAfter,
block.timestamp);

        // update remaining collateral on loan
        _updateLoanCollateral(synthLoan, synthLoan.collateralAmount.sub(totalCollateralLiquidated));

        // Send liquidated BNB collateral to msg.sender
        msg.sender.transfer(totalCollateralLiquidated);

        // emit loan liquidation event
        emit LoanPartiallyLiquidated(
            _loanCreatorsAddress,
            _loanID,
            msg.sender,
            amountToLiquidate,
            totalCollateralLiquidated
        );
    }

    function _splitInterestLoanPayment(
        uint256 _paymentAmount,
        uint256 _accruedInterest,
        uint256 _loanAmount
    )
        internal
        pure
        returns (
            uint256 interestPaid,
            uint256 loanAmountPaid,
            uint256 accruedInterestAfter,
            uint256 loanAmountAfter
        )
    {
        uint256 remainingPayment = _paymentAmount;

        // repay any accrued interests first
        accruedInterestAfter = _accruedInterest;
        if (remainingPayment > 0 && _accruedInterest > 0) {
            // Max repay is the accruedInterest amount
            interestPaid = remainingPayment > _accruedInterest ? _accruedInterest : remainingPayment;
            accruedInterestAfter = accruedInterestAfter.sub(interestPaid);
            remainingPayment = remainingPayment.sub(interestPaid);
        }

        // Remaining amounts - pay down loan amount
        loanAmountAfter = _loanAmount;
        if (remainingPayment > 0) {
            loanAmountAfter = loanAmountAfter.sub(remainingPayment);
            loanAmountPaid = remainingPayment;
        }
    }

    function _processInterestAndLoanPayment(uint256 interestPaid, uint256 loanAmountPaid) internal {
        // Fee distribution. Mint the zUSD fees into the FeePool and record fees paid
        if (interestPaid > 0) {
            synthsUSD().issue(FEE_ADDRESS, interestPaid);
            feePool().recordFeePaid(interestPaid);
        }

        // Decrement totalIssuedSynths
        if (loanAmountPaid > 0) {
            totalIssuedSynths = totalIssuedSynths.sub(loanAmountPaid);
        }
    }

    // Liquidation of an open loan available for anyone
```

```
    function liquidateUnclosedLoan(address _loanCreatorsAddress, uint256 _loanID) external nonReentrant ETHRateNotInvalid {
        require(loanLiquidationOpen, "Liquidation is not open");
        // Close the creators loan and send collateral to the closer.
        _closeLoan(_loanCreatorsAddress, _loanID, true);
        // Tell the Dapps this loan was liquidated
        emit LoanLiquidated(_loanCreatorsAddress, _loanID, msg.sender);
    }

    // ========== PRIVATE FUNCTIONS ==========

    function _closeLoan(
        address account,
        uint256 loanID,
        bool liquidation
    ) private {
        systemStatus().requireIssuanceActive();

        // Get the loan from storage
        SynthLoanStruct memory synthLoan = _getLoanFromStorage(account, loanID);

        // Check loan exists and is open
        _checkLoanIsOpen(synthLoan);

        // Calculate and deduct accrued interest (5%) for fee pool
        // Accrued interests (captured in loanAmount) + new interests
        uint256 interestAmount = accruedInterestOnLoan(
            synthLoan.loanAmount.add(synthLoan.accruedInterest),
            _timeSinceInterestAccrual(synthLoan)
        );
        uint256 repayAmount = synthLoan.loanAmount.add(interestAmount);

        uint256 totalAccruedInterest = synthLoan.accruedInterest.add(interestAmount);

        require(
            IERC20(address(synthsUSD())).balanceOf(msg.sender) >= repayAmount,
            "You do not have the required Zasset balance to close this loan."
        );

        // Record loan as closed
        _recordLoanClosure(synthLoan);

        // Decrement totalIssuedSynths
        // subtract the accrued interest from the loanAmount
        totalIssuedSynths = totalIssuedSynths.sub(synthLoan.loanAmount.sub(synthLoan.accruedInterest));

        // Burn all Synths issued for the loan + the fees
        synthsUSD().burn(msg.sender, repayAmount);

        // Fee distribution. Mint the zUSD fees into the FeePool and record fees paid
        synthsUSD().issue(FEE_ADDRESS, totalAccruedInterest);
        feePool().recordFeePaid(totalAccruedInterest);

        uint256 remainingCollateral = synthLoan.collateralAmount;

        if (liquidation) {
            // Send liquidator redeemed collateral + 10% penalty
            uint256 collateralRedeemed = exchangeRates().effectiveValue(zUSD, repayAmount, COLLATERAL);

            // add penalty
            uint256 totalCollateralLiquidated = collateralRedeemed.multiplyDecimal(
                SafeDecimalMath.unit().add(liquidationPenalty)
            );

            // ensure remaining BNB collateral sufficient to cover collateral liquidated
            // will revert if the liquidated collateral + penalty is more than remaining collateral
            remainingCollateral = remainingCollateral.sub(totalCollateralLiquidated);

            // Send liquidator CollateralLiquidated
            msg.sender.transfer(totalCollateralLiquidated);
        }

        // Send remaining collateral to loan creator
        synthLoan.account.transfer(remainingCollateral);

        // Tell the Dapps
        emit LoanClosed(account, loanID, totalAccruedInterest);
    }

    function _getLoanFromStorage(address account, uint256 loanID) private view returns (SynthLoanStruct memory) {
        SynthLoanStruct[] memory synthLoans = accountsSynthLoans[account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == loanID) {
                return synthLoans[i];
            }
```

```
        }
    }

    function _updateLoan(
        SynthLoanStruct memory _synthLoan,
        uint256 _newLoanAmount,
        uint256 _newAccruedInterest,
        uint256 _lastInterestAccrued
    ) private {
        // Get storage pointer to the accounts array of loans
        SynthLoanStruct[] storage synthLoans = accountsSynthLoans[_synthLoan.account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == _synthLoan.loanID) {
                synthLoans[i].loanAmount = _newLoanAmount;
                synthLoans[i].accruedInterest = _newAccruedInterest;
                synthLoans[i].lastInterestAccrued = uint40(_lastInterestAccrued);
            }
        }
    }

    function _updateLoanCollateral(SynthLoanStruct memory _synthLoan, uint256 _newCollateralAmount)
        private
        returns (SynthLoanStruct memory)
    {
        // Get storage pointer to the accounts array of loans
        SynthLoanStruct[] storage synthLoans = accountsSynthLoans[_synthLoan.account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == _synthLoan.loanID) {
                synthLoans[i].collateralAmount = _newCollateralAmount;
                return synthLoans[i];
            }
        }
    }

    function _recordLoanClosure(SynthLoanStruct memory synthLoan) private {
        // Get storage pointer to the accounts array of loans
        SynthLoanStruct[] storage synthLoans = accountsSynthLoans[synthLoan.account];
        for (uint256 i = 0; i < synthLoans.length; i++) {
            if (synthLoans[i].loanID == synthLoan.loanID) {
                // Record the time the loan was closed
                synthLoans[i].timeClosed = block.timestamp;
            }
        }

        // Reduce Total Open Loans Count
        totalOpenLoanCount = totalOpenLoanCount.sub(1);
    }

    function _incrementTotalLoansCounter() private returns (uint256) {
        // Increase the total Open loan count
        totalOpenLoanCount = totalOpenLoanCount.add(1);
        // Increase the total Loans Created count
        totalLoansCreated = totalLoansCreated.add(1);
        // Return total count to be used as a unique ID.
        return totalLoansCreated;
    }

    function _calculateMintingFee(uint256 _loanAmount) private view returns (uint256 mintingFee) {
        mintingFee = _loanAmount.multiplyDecimalRound(issueFeeRate);
    }

    function _timeSinceInterestAccrual(SynthLoanStruct memory _synthLoan) private view returns (uint256
timeSinceAccrual) {
        // The last interest accrued timestamp for the loan
        // If lastInterestAccrued timestamp is not set (0), use loan timeCreated
        uint256 lastInterestAccrual = _synthLoan.lastInterestAccrued > 0
            ? uint256(_synthLoan.lastInterestAccrued)
            : _synthLoan.timeCreated;

        // diff between last interested accrued and now
        // use loan's timeClosed if loan is closed
        timeSinceAccrual = _synthLoan.timeClosed > 0
            ? _synthLoan.timeClosed.sub(lastInterestAccrual)
            : block.timestamp.sub(lastInterestAccrual);
    }

    function _checkLoanIsOpen(SynthLoanStruct memory _synthLoan) internal pure {
        require(_synthLoan.loanID > 0, "Loan does not exist");
        require(_synthLoan.timeClosed == 0, "Loan already closed");
    }

    /* ========== INTERNAL VIEWS ========== */

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus
address"));
    }
```

```
    function synthsUSD() internal view returns (ISynth) {
        return ISynth(requireAndGetAddress(CONTRACT_ZASSETZUSD, "Missing ZassetzUSD address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    function feePool() internal view returns (IFeePool) {
        return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL, "Missing FeePool address"));
    }

    /* ========== MODIFIERS ========== */

    modifier ETHRateNotInvalid() {
        require(!exchangeRates().rateIsInvalid(COLLATERAL), "Blocked as BNB rate is invalid");
        _;
    }

    // ========== EVENTS ==========

    event CollateralizationRatioUpdated(uint256 ratio);
    event LiquidationRatioUpdated(uint256 ratio);
    event InterestRateUpdated(uint256 interestRate);
    event IssueFeeRateUpdated(uint256 issueFeeRate);
    event IssueLimitUpdated(uint256 issueLimit);
    event MinLoanCollateralSizeUpdated(uint256 minLoanCollateralSize);
    event AccountLoanLimitUpdated(uint256 loanLimit);
    event LoanLiquidationOpenUpdated(bool loanLiquidationOpen);
    event LoanCreated(address indexed account, uint256 loanID, uint256 amount);
    event LoanClosed(address indexed account, uint256 loanID, uint256 feesPaid);
    event LoanLiquidated(address indexed account, uint256 loanID, address liquidator);
    event LoanPartiallyLiquidated(
        address indexed account,
        uint256 loanID,
        address liquidator,
        uint256 liquidatedAmount,
        uint256 liquidatedCollateral
    );
    event CollateralDeposited(address indexed account, uint256 loanID, uint256 collateralAmount, uint256 collateralAfter);
    event CollateralWithdrawn(address indexed account, uint256 loanID, uint256 amountWithdrawn, uint256 collateralAfter);
    event LoanRepaid(address indexed account, uint256 loanID, uint256 repaidAmount, uint256 newLoanAmount);
}
```

Exchanger.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/IExchanger.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IExchangeState.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/IDelegateApprovals.sol";
import "./interfaces/IIssuer.sol";
import "./interfaces/ITradingRewards.sol";
import "./interfaces/IDebtCache.sol";
import "./interfaces/IVirtualSynth.sol";
import "./Proxyable.sol";

// Note: use OZ's IERC20 here as using ours will complain about conflicting names
// during the build (VirtualSynth has IERC20 from the OZ ERC20 implementation)
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/IERC20.sol";


// Used to have strongly-typed access to internal mutative functions in Synthetix
interface ISynthetixInternal {
    function emitExchangeTracking(
        bytes32 trackingCode,
        bytes32 toCurrencyKey,
```

```
        uint256 toAmount
    ) external;

    function emitSynthExchange(
        address account,
        bytes32 fromCurrencyKey,
        uint fromAmount,
        bytes32 toCurrencyKey,
        uint toAmount,
        address toAddress
    ) external;

    function emitExchangeReclaim(
        address account,
        bytes32 currencyKey,
        uint amount
    ) external;

    function emitExchangeRebate(
        address account,
        bytes32 currencyKey,
        uint amount
    ) external;
}


interface IExchangerInternalDebtCache {
    function updateCachedSynthDebtsWithRates(bytes32[] calldata currencyKeys, uint[] calldata currencyRates)
  external;

    function updateCachedSynthDebts(bytes32[] calldata currencyKeys) external;
}


// https://docs.synthetix.io/contracts/source/contracts/exchanger
contract Exchanger is Owned, MixinResolver, MixinSystemSettings, IExchanger {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    struct ExchangeEntrySettlement {
        bytes32 src;
        uint amount;
        bytes32 dest;
        uint reclaim;
        uint rebate;
        uint srcRoundIdAtPeriodEnd;
        uint destRoundIdAtPeriodEnd;
        uint timestamp;
    }

    bytes32 private constant zUSD = "zUSD";

    // SIP-65: Decentralized circuit breaker
    uint public constant CIRCUIT_BREAKER_SUSPENSION_REASON = 65;

    mapping(bytes32 => uint) public lastExchangeRate;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_EXCHANGESTATE = "ExchangeState";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_FEEPOOL = "FeePool";
    bytes32 private constant CONTRACT_TRADING_REWARDS = "TradingRewards";
    bytes32 private constant CONTRACT_DELEGATEAPPROVALS = "DelegateApprovals";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_DEBTCACHE = "DebtCache";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_EXCHANGESTATE,
        CONTRACT_EXRATES,
        CONTRACT_SYNTHETIX,
        CONTRACT_FEEPOOL,
        CONTRACT_TRADING_REWARDS,
        CONTRACT_DELEGATEAPPROVALS,
        CONTRACT_ISSUER,
        CONTRACT_DEBTCACHE
    ];

    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {}
```

```
    /* ========== VIEWS ========== */

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function exchangeState() internal view returns (IExchangeState) {
        return IExchangeState(requireAndGetAddress(CONTRACT_EXCHANGESTATE, "Missing ExchangeState address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function feePool() internal view returns (IFeePool) {
        return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL, "Missing FeePool address"));
    }

    function tradingRewards() internal view returns (ITradingRewards) {
        return ITradingRewards(requireAndGetAddress(CONTRACT_TRADING_REWARDS, "Missing TradingRewards address"));
    }

    function delegateApprovals() internal view returns (IDelegateApprovals) {
        return IDelegateApprovals(requireAndGetAddress(CONTRACT_DELEGATEAPPROVALS, "Missing DelegateApprovals address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function debtCache() internal view returns (IExchangerInternalDebtCache) {
        return IExchangerInternalDebtCache(requireAndGetAddress(CONTRACT_DEBTCACHE, "Missing DebtCache address"));
    }

    function maxSecsLeftInWaitingPeriod(address account, bytes32 currencyKey) public view returns (uint) {
        return secsLeftInWaitingPeriodForExchange(exchangeState().getMaxTimestamp(account, currencyKey));
    }

    function waitingPeriodSecs() external view returns (uint) {
        return getWaitingPeriodSecs();
    }

    function tradingRewardsEnabled() external view returns (bool) {
        return getTradingRewardsEnabled();
    }

    function priceDeviationThresholdFactor() external view returns (uint) {
        return getPriceDeviationThresholdFactor();
    }

    function settlementOwing(address account, bytes32 currencyKey)
        public
        view
        returns (
            uint reclaimAmount,
            uint rebateAmount,
            uint numEntries
        )
    {
        (reclaimAmount, rebateAmount, numEntries, ) = _settlementOwing(account, currencyKey);
    }

    // Internal function to emit events for each individual rebate and reclaim entry
    function _settlementOwing(address account, bytes32 currencyKey)
        internal
        view
        returns (
            uint reclaimAmount,
            uint rebateAmount,
            uint numEntries,
            ExchangeEntrySettlement[] memory
        )
    {
        // Need to sum up all reclaim and rebate amounts for the user and the currency key
        numEntries = exchangeState().getLengthOfEntries(account, currencyKey);
```

```
        // For each unsettled exchange
        ExchangeEntrySettlement[] memory settlements = new ExchangeEntrySettlement[](numEntries);
        for (uint i = 0; i < numEntries; i++) {
            uint reclaim;
            uint rebate;
            // fetch the entry from storage
            IExchangeState.ExchangeEntry memory exchangeEntry = _getExchangeEntry(account, currencyKey, i);

            // determine the last round ids for src and dest pairs when period ended or latest if not over
            (uint srcRoundIdAtPeriodEnd, uint destRoundIdAtPeriodEnd) = getRoundIdsAtPeriodEnd(exchangeEntry);

            // given these round ids, determine what effective value they should have received
            uint destinationAmount = exchangeRates().effectiveValueAtRound(
                exchangeEntry.src,
                exchangeEntry.amount,
                exchangeEntry.dest,
                srcRoundIdAtPeriodEnd,
                destRoundIdAtPeriodEnd
            );

            // and deduct the fee from this amount using the exchangeFeeRate from storage
            uint amountShouldHaveReceived = _getAmountReceivedForExchange(destinationAmount, exchangeEntry.exchangeFeeRate);

            // SIP-65 settlements where the amount at end of waiting period is beyond the threshold, then
            // settle with no reclaim or rebate
            if (!_isDeviationAboveThreshold(exchangeEntry.amountReceived, amountShouldHaveReceived)) {
                if (exchangeEntry.amountReceived > amountShouldHaveReceived) {
                    // if they received more than they should have, add to the reclaim tally
                    reclaim = exchangeEntry.amountReceived.sub(amountShouldHaveReceived);
                    reclaimAmount = reclaimAmount.add(reclaim);
                } else if (amountShouldHaveReceived > exchangeEntry.amountReceived) {
                    // if less, add to the rebate tally
                    rebate = amountShouldHaveReceived.sub(exchangeEntry.amountReceived);
                    rebateAmount = rebateAmount.add(rebate);
                }
            }

            settlements[i] = ExchangeEntrySettlement({
                src: exchangeEntry.src,
                amount: exchangeEntry.amount,
                dest: exchangeEntry.dest,
                reclaim: reclaim,
                rebate: rebate,
                srcRoundIdAtPeriodEnd: srcRoundIdAtPeriodEnd,
                destRoundIdAtPeriodEnd: destRoundIdAtPeriodEnd,
                timestamp: exchangeEntry.timestamp
            });
        }

        return (reclaimAmount, rebateAmount, numEntries, settlements);
    }

    function _getExchangeEntry(
        address account,
        bytes32 currencyKey,
        uint index
    ) internal view returns (IExchangeState.ExchangeEntry memory) {
        (
            bytes32 src,
            uint amount,
            bytes32 dest,
            uint amountReceived,
            uint exchangeFeeRate,
            uint timestamp,
            uint roundIdForSrc,
            uint roundIdForDest
        ) = exchangeState().getEntryAt(account, currencyKey, index);

        return
            IExchangeState.ExchangeEntry({
                src: src,
                amount: amount,
                dest: dest,
                amountReceived: amountReceived,
                exchangeFeeRate: exchangeFeeRate,
                timestamp: timestamp,
                roundIdForSrc: roundIdForSrc,
                roundIdForDest: roundIdForDest
            });
    }

    function hasWaitingPeriodOrSettlementOwing(address account, bytes32 currencyKey) external view returns (bool) {
```

```
        if (maxSecsLeftInWaitingPeriod(account, currencyKey) != 0) {
            return true;
        }

        (uint reclaimAmount, , , ) = _settlementOwing(account, currencyKey);

        return reclaimAmount > 0;
    }

    /* ========== SETTERS ========== */

    function calculateAmountAfterSettlement(
        address from,
        bytes32 currencyKey,
        uint amount,
        uint refunded
    ) public view returns (uint amountAfterSettlement) {
        amountAfterSettlement = amount;

        // balance of a synth will show an amount after settlement
        uint balanceOfSourceAfterSettlement =
IERC20(address(issuer().synths(currencyKey))).balanceOf(from);

        // when there isn't enough supply (either due to reclamation settlement or because the number is too high)
        if (amountAfterSettlement > balanceOfSourceAfterSettlement) {
            // then the amount to exchange is reduced to their remaining supply
            amountAfterSettlement = balanceOfSourceAfterSettlement;
        }

        if (refunded > 0) {
            amountAfterSettlement = amountAfterSettlement.add(refunded);
        }
    }

    function isSynthRateInvalid(bytes32 currencyKey) external view returns (bool) {
        return _isSynthRateInvalid(currencyKey, exchangeRates().rateForCurrency(currencyKey));
    }

    /* ========== MUTATIVE FUNCTIONS ========== */
    function exchange(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress
    ) external onlySynthetixorSynth returns (uint amountReceived) {
        uint fee;
        (amountReceived, fee, ) = _exchange(
            from,
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey,
            destinationAddress,
            false
        );

        _processTradingRewards(fee, destinationAddress);
    }

    function exchangeOnBehalf(
        address exchangeForAddress,
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external onlySynthetixorSynth returns (uint amountReceived) {
        require(delegateApprovals().canExchangeFor(exchangeForAddress, from), "Not approved to act on
behalf");

        uint fee;
        (amountReceived, fee, ) = _exchange(
            exchangeForAddress,
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey,
            exchangeForAddress,
            false
        );

        _processTradingRewards(fee, exchangeForAddress);
    }

    function exchangeWithTracking(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
```

```
        address destinationAddress,
        address originator,
        bytes32 trackingCode
    ) external onlySynthetixorSynth returns (uint amountReceived) {
        uint fee;
        (amountReceived, fee, ) = _exchange(
            from,
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey,
            destinationAddress,
            false
        );

        _processTradingRewards(fee, originator);

        _emitTrackingEvent(trackingCode, destinationCurrencyKey, amountReceived);
    }

    function exchangeOnBehalfWithTracking(
        address exchangeForAddress,
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external onlySynthetixorSynth returns (uint amountReceived) {
        require(delegateApprovals().canExchangeFor(exchangeForAddress, from), "Not approved to act on behalf");

        uint fee;
        (amountReceived, fee, ) = _exchange(
            exchangeForAddress,
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey,
            exchangeForAddress,
            false
        );

        _processTradingRewards(fee, originator);

        _emitTrackingEvent(trackingCode, destinationCurrencyKey, amountReceived);
    }

    function exchangeWithVirtual(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress,
        bytes32 trackingCode
    ) external onlySynthetixorSynth returns (uint amountReceived, IVirtualSynth vSynth) {
        uint fee;
        (amountReceived, fee, vSynth) = _exchange(
            from,
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey,
            destinationAddress,
            true
        );

        _processTradingRewards(fee, destinationAddress);

        if (trackingCode != bytes32(0)) {
            _emitTrackingEvent(trackingCode, destinationCurrencyKey, amountReceived);
        }
    }

    function _emitTrackingEvent(
        bytes32 trackingCode,
        bytes32 toCurrencyKey,
        uint256 toAmount
    ) internal {
        ISynthetixInternal(address(synthetix())).emitExchangeTracking(trackingCode, toCurrencyKey, toAmount);
    }

    function _processTradingRewards(uint fee, address originator) internal {
        if (fee > 0 && originator != address(0) && getTradingRewardsEnabled()) {
            tradingRewards().recordExchangeFeeForAccount(fee, originator);
        }
    }

    function _suspendIfRateInvalid(bytes32 currencyKey, uint rate) internal returns (bool circuitBroken) {
```

```
        if (_isSynthRateInvalid(currencyKey, rate)) {
            systemStatus().suspendSynth(currencyKey, CIRCUIT_BREAKER_SUSPENSION_REASON);
            circuitBroken = true;
        } else {
            lastExchangeRate[currencyKey] = rate;
        }
    }

    function _updateSNXIssuedDebtOnExchange(bytes32[2] memory currencyKeys, uint[2] memory currencyRates) internal {
        bool includeszUSD = currencyKeys[0] == zUSD || currencyKeys[1] == zUSD;
        uint numKeys = includeszUSD ? 2 : 3;

        bytes32[] memory keys = new bytes32[](numKeys);
        keys[0] = currencyKeys[0];
        keys[1] = currencyKeys[1];

        uint[] memory rates = new uint[](numKeys);
        rates[0] = currencyRates[0];
        rates[1] = currencyRates[1];

        if (!includeszUSD) {
            keys[2] = zUSD; // And we'll also update zUSD to account for any fees if it wasn't one of the exchanged currencies
            rates[2] = SafeDecimalMath.unit();
        }

        // Note that exchanges can't invalidate the debt cache, since if a rate is invalid,
        // the exchange will have failed already.
        debtCache().updateCachedSynthDebtsWithRates(keys, rates);
    }

    function _settleAndCalcSourceAmountRemaining(
        uint sourceAmount,
        address from,
        bytes32 sourceCurrencyKey
    ) internal returns (uint sourceAmountAfterSettlement) {
        (, uint refunded, uint numEntriesSettled) = _internalSettle(from, sourceCurrencyKey, false);

        sourceAmountAfterSettlement = sourceAmount;

        // when settlement was required
        if (numEntriesSettled > 0) {
            // ensure the sourceAmount takes this into account
            sourceAmountAfterSettlement = calculateAmountAfterSettlement(from, sourceCurrencyKey, sourceAmount, refunded);
        }
    }

    function _exchange(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress,
        bool virtualSynth
    )
        internal
        returns (
            uint amountReceived,
            uint fee,
            IVirtualSynth vSynth
        )
    {
        _ensureCanExchange(sourceCurrencyKey, sourceAmount, destinationCurrencyKey);

        uint sourceAmountAfterSettlement = _settleAndCalcSourceAmountRemaining(sourceAmount, from, sourceCurrencyKey);

        // If, after settlement the user has no balance left (highly unlikely), then return to prevent
        // emitting events of 0 and don't revert so as to ensure the settlement queue is emptied
        if (sourceAmountAfterSettlement == 0) {
            return (0, 0, IVirtualSynth(0));
        }

        uint exchangeFeeRate;
        uint sourceRate;
        uint destinationRate;

        // Note: `fee` is denominated in the destinationCurrencyKey.
        (amountReceived, fee, exchangeFeeRate, sourceRate, destinationRate) = _getAmountsForExchangeMinusFees(
            sourceAmountAfterSettlement,
            sourceCurrencyKey,
            destinationCurrencyKey
        );
```

```
        // SIP-65: Decentralized Circuit Breaker
        if (
            _suspendIfRateInvalid(sourceCurrencyKey, sourceRate) ||
            _suspendIfRateInvalid(destinationCurrencyKey, destinationRate)
        ) {
            return (0, 0, IVirtualSynth(0));
        }

        // Note: We don't need to check their balance as the burn() below will do a safe subtraction which requires
        // the subtraction to not overflow, which would happen if their balance is not sufficient.

        vSynth = _convert(
            sourceCurrencyKey,
            from,
            sourceAmountAfterSettlement,
            destinationCurrencyKey,
            amountReceived,
            destinationAddress,
            virtualSynth
        );

        // When using a virtual synth, it becomes the destinationAddress for event and settlement tracking
        if (vSynth != IVirtualSynth(0)) {
            destinationAddress = address(vSynth);
        }

        // Remit the fee if required
        if (fee > 0) {
            // Normalize fee to zUSD
            // Note: `fee` is being reused to avoid stack too deep errors.
            fee = exchangeRates().effectiveValue(destinationCurrencyKey, fee, zUSD);

            // Remit the fee in zUSDs
            issuer().synths(zUSD).issue(feePool().FEE_ADDRESS(), fee);

            // Tell the fee pool about this
            feePool().recordFeePaid(fee);
        }

        // Note: As of this point, `fee` is denominated in zUSD.

        // Nothing changes as far as issuance data goes because the total value in the system hasn't changed.
        // But we will update the debt snapshot in case exchange rates have fluctuated since the last exchange
        // in these currencies
        _updateSNXIssuedDebtOnExchange([sourceCurrencyKey, destinationCurrencyKey], [sourceRate,
destinationRate]);

        // Let the DApps know there was a Synth exchange
        ISynthetixInternal(address(synthetix())).emitSynthExchange(
            from,
            sourceCurrencyKey,
            sourceAmountAfterSettlement,
            destinationCurrencyKey,
            amountReceived,
            destinationAddress
        );

        // persist the exchange information for the dest key
        appendExchange(
            destinationAddress,
            sourceCurrencyKey,
            sourceAmountAfterSettlement,
            destinationCurrencyKey,
            amountReceived,
            exchangeFeeRate
        );
    }

    function _convert(
        bytes32 sourceCurrencyKey,
        address from,
        uint sourceAmountAfterSettlement,
        bytes32 destinationCurrencyKey,
        uint amountReceived,
        address recipient,
        bool virtualSynth
    ) internal returns (IVirtualSynth vSynth) {
        // Burn the source amount
        issuer().synths(sourceCurrencyKey).burn(from, sourceAmountAfterSettlement);

        // Issue their new synths
        ISynth dest = issuer().synths(destinationCurrencyKey);

        if (virtualSynth) {
            Proxyable synth = Proxyable(address(dest));
            vSynth = _createVirtualSynth(IERC20(address(synth.proxy())), recipient, amountReceived,
destinationCurrencyKey);
```

```
            dest.issue(address(vSynth), amountReceived);
        } else {
            dest.issue(recipient, amountReceived);
        }
    }

    function _createVirtualSynth(
        IERC20,
        address,
        uint,
        bytes32
    ) internal returns (IVirtualSynth) {
        revert("Cannot be run on this layer");
    }

    // Note: this function can intentionally be called by anyone on behalf of anyone else (the caller just pays the gas)
    function settle(address from, bytes32 currencyKey)
        external
        returns (
            uint reclaimed,
            uint refunded,
            uint numEntriesSettled
        )
    {
        systemStatus().requireSynthActive(currencyKey);
        return _internalSettle(from, currencyKey, true);
    }

    function suspendSynthWithInvalidRate(bytes32 currencyKey) external {
        systemStatus().requireSystemActive();
        require(issuer().synths(currencyKey) != ISynth(0), "No such hasset");
        require(_isSynthRateInvalid(currencyKey, exchangeRates().rateForCurrency(currencyKey)), "Hasset price is valid");
        systemStatus().suspendSynth(currencyKey, CIRCUIT_BREAKER_SUSPENSION_REASON);
    }

    // SIP-78
    function setLastExchangeRateForSynth(bytes32 currencyKey, uint rate) external onlyExchangeRates {
        require(rate > 0, "Rate must be above 0");
        lastExchangeRate[currencyKey] = rate;
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _ensureCanExchange(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) internal view {
        require(sourceCurrencyKey != destinationCurrencyKey, "Can't be same hasset");
        require(sourceAmount > 0, "Zero amount");

        bytes32[] memory synthKeys = new bytes32[](2);
        synthKeys[0] = sourceCurrencyKey;
        synthKeys[1] = destinationCurrencyKey;
        require(!exchangeRates().anyRateIsInvalid(synthKeys), "Src/dest rate invalid or not found");
    }

    function _isSynthRateInvalid(bytes32 currencyKey, uint currentRate) internal view returns (bool) {
        if (currentRate == 0) {
            return true;
        }

        uint lastRateFromExchange = lastExchangeRate[currencyKey];

        if (lastRateFromExchange > 0) {
            return _isDeviationAboveThreshold(lastRateFromExchange, currentRate);
        }

        // if no last exchange for this synth, then we need to look up last 3 rates (+1 for current rate)
        (uint[] memory rates, ) = exchangeRates().ratesAndUpdatedTimeForCurrencyLastNRounds(currencyKey, 4);

        // start at index 1 to ignore current rate
        for (uint i = 1; i < rates.length; i++) {
            // ignore any empty rates in the past (otherwise we will never be able to get validity)
            if (rates[i] > 0 && _isDeviationAboveThreshold(rates[i], currentRate)) {
                return true;
            }
        }

        return false;
    }

    function _isDeviationAboveThreshold(uint base, uint comparison) internal view returns (bool) {
        if (base == 0 || comparison == 0) {
```

```
            return true;
        }

        uint factor;
        if (comparison > base) {
            factor = comparison.divideDecimal(base);
        } else {
            factor = base.divideDecimal(comparison);
        }

        return factor >= getPriceDeviationThresholdFactor();
    }

    function _internalSettle(
        address from,
        bytes32 currencyKey,
        bool updateCache
    )
        internal
        returns (
            uint reclaimed,
            uint refunded,
            uint numEntriesSettled
        )
    {
        require(maxSecsLeftInWaitingPeriod(from, currencyKey) == 0, "Cannot settle during waiting period");

        (
            uint reclaimAmount,
            uint rebateAmount,
            uint entries,
            ExchangeEntrySettlement[] memory settlements
        ) = _settlementOwing(from, currencyKey);

        if (reclaimAmount > rebateAmount) {
            reclaimed = reclaimAmount.sub(rebateAmount);
            reclaim(from, currencyKey, reclaimed);
        } else if (rebateAmount > reclaimAmount) {
            refunded = rebateAmount.sub(reclaimAmount);
            refund(from, currencyKey, refunded);
        }

        if (updateCache) {
            bytes32[] memory key = new bytes32[](1);
            key[0] = currencyKey;
            debtCache().updateCachedSynthDebts(key);
        }

        // emit settlement event for each settled exchange entry
        for (uint i = 0; i < settlements.length; i++) {
            emit ExchangeEntrySettled(
                from,
                settlements[i].src,
                settlements[i].amount,
                settlements[i].dest,
                settlements[i].reclaim,
                settlements[i].rebate,
                settlements[i].srcRoundIdAtPeriodEnd,
                settlements[i].destRoundIdAtPeriodEnd,
                settlements[i].timestamp
            );
        }

        numEntriesSettled = entries;

        // Now remove all entries, even if no reclaim and no rebate
        exchangeState().removeEntries(from, currencyKey);
    }

    function reclaim(
        address from,
        bytes32 currencyKey,
        uint amount
    ) internal {
        // burn amount from user
        issuer().synths(currencyKey).burn(from, amount);
        ISynthetixInternal(address(synthetix())).emitExchangeReclaim(from, currencyKey, amount);
    }

    function refund(
        address from,
        bytes32 currencyKey,
        uint amount
    ) internal {
        // issue amount to user
        issuer().synths(currencyKey).issue(from, amount);
        ISynthetixInternal(address(synthetix())).emitExchangeRebate(from, currencyKey, amount);
```

```
    }

    function secsLeftInWaitingPeriodForExchange(uint timestamp) internal view returns (uint) {
        uint _waitingPeriodSecs = getWaitingPeriodSecs();
        if (timestamp == 0 || now >= timestamp.add(_waitingPeriodSecs)) {
            return 0;
        }

        return timestamp.add(_waitingPeriodSecs).sub(now);
    }

    function feeRateForExchange(bytes32 sourceCurrencyKey, bytes32 destinationCurrencyKey)
        external
        view
        returns (uint exchangeFeeRate)
    {
        exchangeFeeRate = _feeRateForExchange(sourceCurrencyKey, destinationCurrencyKey);
    }

    function _feeRateForExchange(
        bytes32, // API for source in case pricing model evolves to include source rate /* sourceCurrencyKey */
        bytes32 destinationCurrencyKey
    ) internal view returns (uint exchangeFeeRate) {
        return getExchangeFeeRate(destinationCurrencyKey);
    }

    function getAmountsForExchange(
        uint sourceAmount,
        bytes32 sourceCurrencyKey,
        bytes32 destinationCurrencyKey
    )
        external
        view
        returns (
            uint amountReceived,
            uint fee,
            uint exchangeFeeRate
        )
    {
        (amountReceived, fee, exchangeFeeRate, , ) = _getAmountsForExchangeMinusFees(
            sourceAmount,
            sourceCurrencyKey,
            destinationCurrencyKey
        );
    }

    function _getAmountsForExchangeMinusFees(
        uint sourceAmount,
        bytes32 sourceCurrencyKey,
        bytes32 destinationCurrencyKey
    )
        internal
        view
        returns (
            uint amountReceived,
            uint fee,
            uint exchangeFeeRate,
            uint sourceRate,
            uint destinationRate
        )
    {
        uint destinationAmount;
        (destinationAmount, sourceRate, destinationRate) = exchangeRates().effectiveValueAndRates(
            sourceCurrencyKey,
            sourceAmount,
            destinationCurrencyKey
        );
        exchangeFeeRate = _feeRateForExchange(sourceCurrencyKey, destinationCurrencyKey);
        amountReceived = _getAmountReceivedForExchange(destinationAmount, exchangeFeeRate);
        fee = destinationAmount.sub(amountReceived);
    }

    function _getAmountReceivedForExchange(uint destinationAmount, uint exchangeFeeRate)
        internal
        pure
        returns (uint amountReceived)
    {
        amountReceived                                                                    =
destinationAmount.multiplyDecimal(SafeDecimalMath.unit().sub(exchangeFeeRate));
    }

    function appendExchange(
        address account,
        bytes32 src,
        uint amount,
        bytes32 dest,
        uint amountReceived,
```

```
        uint exchangeFeeRate
    ) internal {
        IExchangeRates exRates = exchangeRates();
        uint roundIdForSrc = exRates.getCurrentRoundId(src);
        uint roundIdForDest = exRates.getCurrentRoundId(dest);
        exchangeState().appendExchangeEntry(
            account,
            src,
            amount,
            dest,
            amountReceived,
            exchangeFeeRate,
            now,
            roundIdForSrc,
            roundIdForDest
        );

        emit ExchangeEntryAppended(
            account,
            src,
            amount,
            dest,
            amountReceived,
            exchangeFeeRate,
            roundIdForSrc,
            roundIdForDest
        );
    }

    function getRoundIdsAtPeriodEnd(IExchangeState.ExchangeEntry memory exchangeEntry)
        internal
        view
        returns (uint srcRoundIdAtPeriodEnd, uint destRoundIdAtPeriodEnd)
    {
        IExchangeRates exRates = exchangeRates();
        uint _waitingPeriodSecs = getWaitingPeriodSecs();

        srcRoundIdAtPeriodEnd = exRates.getLastRoundIdBeforeElapsedSecs(
            exchangeEntry.src,
            exchangeEntry.roundIdForSrc,
            exchangeEntry.timestamp,
            _waitingPeriodSecs
        );
        destRoundIdAtPeriodEnd = exRates.getLastRoundIdBeforeElapsedSecs(
            exchangeEntry.dest,
            exchangeEntry.roundIdForDest,
            exchangeEntry.timestamp,
            _waitingPeriodSecs
        );
    }

    // ========== MODIFIERS ==========

    modifier onlySynthetixorSynth() {
        ISynthetix _synthetix = synthetix();
        require(
            msg.sender == address(_synthetix) || _synthetix.synthsByAddress(msg.sender) != bytes32(0),
            "Exchanger: Only horizon or a hasset contract can perform this action"
        );
        _;
    }

    modifier onlyExchangeRates() {
        IExchangeRates _exchangeRates = exchangeRates();
        require(msg.sender == address(_exchangeRates), "Restricted to ExchangeRates");
        _;
    }

    // ========== EVENTS ==========
    event ExchangeEntryAppended(
        address indexed account,
        bytes32 src,
        uint256 amount,
        bytes32 dest,
        uint256 amountReceived,
        uint256 exchangeFeeRate,
        uint256 roundIdForSrc,
        uint256 roundIdForDest
    );

    event ExchangeEntrySettled(
        address indexed from,
        bytes32 src,
        uint256 amount,
        bytes32 dest,
        uint256 reclaim,
        uint256 rebate,
```

```
            uint256 srcRoundIdAtPeriodEnd,
            uint256 destRoundIdAtPeriodEnd,
            uint256 exchangeTimestamp
        );
}
```

ExchangeRates.sol

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

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/IExchangeRates.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
// AggregatorInterface from Chainlink represents a decentralized pricing network for a single currency key
import "@chainlink/contracts-0.0.10/src/v0.5/interfaces/AggregatorV2V3Interface.sol";
// FlagsInterface from Chainlink addresses SIP-76
import "@chainlink/contracts-0.0.10/src/v0.5/interfaces/FlagsInterface.sol";
import "./interfaces/IExchanger.sol";


interface IStdReference {/// knownsec Reference 结构体返回接口
    // A structure returned whenever someone requests for standard reference data.
    struct ReferenceData {/// knownsec ReferenceData 结构体
        uint256 rate; // base/quote exchange rat00e, multiplied by 1e18.
        uint256 lastUpdatedBase; // UNIX epoch of the last time when base price gets updated.
        uint256 lastUpdatedQuote; // UNIX epoch of the last time when quote price gets updated.
    }

    // Returns the price data for the given base/quote pair. Revert if not available.
    function getReferenceData(string calldata _base, string calldata _quote) external view returns (ReferenceData
  memory);

    // Similar to getReferenceData, but with multiple base/quote pairs at once.
    function getReferenceDataBulk(string[] calldata _bases, string[] calldata _quotes)
        external
        view
        returns (ReferenceData[] memory);// knownsec 外部使用 Reference 数组返回
}


// https://docs.synthetix.io/contracts/source/contracts/exchangerates
contract ExchangeRates is Owned, MixinResolver, MixinSystemSettings, IExchangeRates {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    // Exchange rates and update times stored by currency code, e.g. 'HZN', or 'zUSD'
    mapping(bytes32 => mapping(uint => RateAndUpdatedTime)) private _rates;

    // The address of the oracle which pushes rate updates to this contract
    address public oracle;

    // The address of the BandProtocol address
    IStdReference public bandProtocolOracle;// knownsec Band 协议预言机地址

    // Decentralized oracle networks that feed into pricing aggregators
    mapping(bytes32 => AggregatorV2V3Interface) public aggregators;

    mapping(bytes32 => uint8) public currencyKeyDecimals;

    // List of aggregator keys for convenient iteration
    bytes32[] public aggregatorKeys;

    // Do not allow the oracle to submit times any further forward into the future than this constant.
    uint private constant ORACLE_FUTURE_LIMIT = 10 minutes;

    mapping(bytes32 => InversePricing) public inversePricing;

    bytes32[] public invertedKeys;

    mapping(bytes32 => uint) public currentRoundForRate;

    mapping(bytes32 => uint) public roundFrozen;

    /* ========== ENCODED NAMES ========== */
    bytes32 private constant HZN = "HZN";
    // TODO use SNX as HZN's price at testnet
    bytes32 private constant SNX = "SNX";
```

```
    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";

    bytes32[24] private addressesToCache = [CONTRACT_EXCHANGER];

    //
    // ========== CONSTRUCTOR ==========

    constructor(
        address _owner,
        address _oracle,
        address _resolver,
        bytes32[] memory _currencyKeys,
        uint[] memory _newRates
    ) public Owned(_owner) MixinResolver(_resolver, addressesToCache) MixinSystemSettings() {
        require(_currencyKeys.length == _newRates.length, "Currency key length and rate length must match.");

        oracle = _oracle;

        // The zUSD rate is always 1 and is never stale.
        _setRate("zUSD", SafeDecimalMath.unit(), now);

        internalUpdateRates(_currencyKeys, _newRates, now);
    }

    /* ========== SETTERS ========== */

    function setOracle(address _oracle) external onlyOwner {
        oracle = _oracle;
        emit OracleUpdated(oracle);
    }

    function setBandProtocolOracle(IStdReference _bandProtocolOracle) external onlyOwner {// knownsec Band
预言机地址设置 owner 可用
        bandProtocolOracle = _bandProtocolOracle;
        emit BandProtocolOracleUpdated(bandProtocolOracle);// knownsec 事件记录
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function updateRates(
        bytes32[] calldata currencyKeys,
        uint[] calldata newRates,
        uint timeSent
    ) external onlyOracle returns (bool) {
        return internalUpdateRates(currencyKeys, newRates, timeSent);
    }

    function deleteRate(bytes32 currencyKey) external onlyOracle {
        require(_getRate(currencyKey) > 0, "Rate is zero");

        delete _rates[currencyKey][currentRoundForRate[currencyKey]];

        currentRoundForRate[currencyKey]--;

        emit RateDeleted(currencyKey);
    }

    function setInversePricing(
        bytes32 currencyKey,
        uint entryPoint,
        uint upperLimit,
        uint lowerLimit,
        bool freezeAtUpperLimit,
        bool freezeAtLowerLimit
    ) external onlyOwner {
        // 0 < lowerLimit < entryPoint => 0 < entryPoint
        require(lowerLimit > 0, "lowerLimit must be above 0");
        require(upperLimit > entryPoint, "upperLimit must be above the entryPoint");
        require(upperLimit < entryPoint.mul(2), "upperLimit must be less than double entryPoint");
        require(lowerLimit < entryPoint, "lowerLimit must be below the entryPoint");

        require(!(freezeAtUpperLimit && freezeAtLowerLimit), "Cannot freeze at both limits");

        InversePricing storage inverse = inversePricing[currencyKey];
        if (inverse.entryPoint == 0) {
            // then we are adding a new inverse pricing, so add this
            invertedKeys.push(currencyKey);
        }
        inverse.entryPoint = entryPoint;
        inverse.upperLimit = upperLimit;
        inverse.lowerLimit = lowerLimit;

        if (freezeAtUpperLimit || freezeAtLowerLimit) {
            // When indicating to freeze, we need to know the rate to freeze it at - either upper or lower
            // this is useful in situations where ExchangeRates is updated and there are existing inverted
            // rates already frozen in the current contract that need persisting across the upgrade
```

```
            inverse.frozenAtUpperLimit = freezeAtUpperLimit;
            inverse.frozenAtLowerLimit = freezeAtLowerLimit;
            uint roundId = _getCurrentRoundId(currencyKey);
            roundFrozen[currencyKey] = roundId;
            emit InversePriceFrozen(currencyKey, freezeAtUpperLimit ? upperLimit : lowerLimit, roundId,
msg.sender);
        } else {
            // unfreeze if need be
            inverse.frozenAtUpperLimit = false;
            inverse.frozenAtLowerLimit = false;
            // remove any tracking
            roundFrozen[currencyKey] = 0;
        }

        // SIP-78
        uint rate = _getRate(currencyKey);
        if (rate > 0) {
            exchanger().setLastExchangeRateForSynth(currencyKey, rate);
        }

        emit InversePriceConfigured(currencyKey, entryPoint, upperLimit, lowerLimit);
    }

    function removeInversePricing(bytes32 currencyKey) external onlyOwner {
        require(inversePricing[currencyKey].entryPoint > 0, "No inverted price exists");

        delete inversePricing[currencyKey];

        // now remove inverted key from array
        bool wasRemoved = removeFromArray(currencyKey, invertedKeys);

        if (wasRemoved) {
            emit InversePriceConfigured(currencyKey, 0, 0, 0);
        }
    }

    function addAggregator(bytes32 currencyKey, address aggregatorAddress) external onlyOwner {
        AggregatorV2V3Interface aggregator = AggregatorV2V3Interface(aggregatorAddress);
        // This check tries to make sure that a valid aggregator is being added.
        // It checks if the aggregator is an existing smart contract that has implemented `latestTimestamp` function.

        require(aggregator.latestRound() >= 0, "Given Aggregator is invalid");
        uint8 decimals = aggregator.decimals();
        require(decimals <= 18, "Aggregator decimals should be lower or equal to 18");
        if (address(aggregators[currencyKey]) == address(0)) {
            aggregatorKeys.push(currencyKey);
        }
        aggregators[currencyKey] = aggregator;
        currencyKeyDecimals[currencyKey] = decimals;
        emit AggregatorAdded(currencyKey, address(aggregator));
    }

    function removeAggregator(bytes32 currencyKey) external onlyOwner {
        address aggregator = address(aggregators[currencyKey]);
        require(aggregator != address(0), "No aggregator exists for key");
        delete aggregators[currencyKey];
        delete currencyKeyDecimals[currencyKey];

        bool wasRemoved = removeFromArray(currencyKey, aggregatorKeys);

        if (wasRemoved) {
            emit AggregatorRemoved(currencyKey, aggregator);
        }
    }

    // SIP-75 Public keeper function to freeze a synth that is out of bounds
    function freezeRate(bytes32 currencyKey) external {
        InversePricing storage inverse = inversePricing[currencyKey];
        require(inverse.entryPoint > 0, "Cannot freeze non-inverse rate");
        require(!inverse.frozenAtUpperLimit && !inverse.frozenAtLowerLimit, "The rate is already frozen");

        uint rate = _getRate(currencyKey);

        if (rate > 0 && (rate >= inverse.upperLimit || rate <= inverse.lowerLimit)) {
            inverse.frozenAtUpperLimit = (rate == inverse.upperLimit);
            inverse.frozenAtLowerLimit = (rate == inverse.lowerLimit);
            uint currentRoundId = _getCurrentRoundId(currencyKey);
            roundFrozen[currencyKey] = currentRoundId;
            emit InversePriceFrozen(currencyKey, rate, currentRoundId, msg.sender);
        } else {
            revert("Rate within bounds");
        }
    }

    /* ========== VIEWS ========== */
```

```
    // SIP-75 View to determine if freezeRate can be called safely
    function canFreezeRate(bytes32 currencyKey) external view returns (bool) {
        InversePricing memory inverse = inversePricing[currencyKey];
        if (inverse.entryPoint == 0 || inverse.frozenAtUpperLimit || inverse.frozenAtLowerLimit) {
            return false;
        } else {
            uint rate = _getRate(currencyKey);
            return (rate > 0 && (rate >= inverse.upperLimit || rate <= inverse.lowerLimit));
        }
    }

    function currenciesUsingAggregator(address aggregator) external view returns (bytes32[] memory currencies)
{
        uint count = 0;
        currencies = new bytes32[](aggregatorKeys.length);
        for (uint i = 0; i < aggregatorKeys.length; i++) {
            bytes32 currencyKey = aggregatorKeys[i];
            if (address(aggregators[currencyKey]) == aggregator) {
                currencies[count++] = currencyKey;
            }
        }
    }

    function rateStalePeriod() external view returns (uint) {
        return getRateStalePeriod();
    }

    function aggregatorWarningFlags() external view returns (address) {
        return getAggregatorWarningFlags();
    }

    function rateAndUpdatedTime(bytes32 currencyKey) external view returns (uint rate, uint time) {
        RateAndUpdatedTime memory rateAndTime = _getRateAndUpdatedTime(currencyKey);
        return (rateAndTime.rate, rateAndTime.time);
    }

    function getLastRoundIdBeforeElapsedSecs(
        bytes32 currencyKey,
        uint startingRoundId,
        uint startingTimestamp,
        uint timediff
    ) external view returns (uint) {
        uint roundId = startingRoundId;
        uint nextTimestamp = 0;
        while (true) {
            (, nextTimestamp) = _getRateAndTimestampAtRound(currencyKey, roundId + 1);
            // if there's no new round, then the previous roundId was the latest
            if (nextTimestamp == 0 || nextTimestamp > startingTimestamp + timediff) {
                return roundId;
            }
            roundId++;
        }
        return roundId;
    }

    function getCurrentRoundId(bytes32 currencyKey) external view returns (uint) {
        return _getCurrentRoundId(currencyKey);
    }

    function effectiveValueAtRound(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        uint roundIdForSrc,
        uint roundIdForDest
    ) external view returns (uint value) {
        // If there's no change in the currency, then just return the amount they gave us
        if (sourceCurrencyKey == destinationCurrencyKey) return sourceAmount;

        (uint srcRate, ) = _getRateAndTimestampAtRound(sourceCurrencyKey, roundIdForSrc);
        (uint destRate, ) = _getRateAndTimestampAtRound(destinationCurrencyKey, roundIdForDest);
        if (destRate == 0) {
            // prevent divide-by 0 error (this can happen when roundIDs jump epochs due
            // to aggregator upgrades)
            return 0;
        }
        // Calculate the effective value by going from source -> USD -> destination
        value = sourceAmount.multiplyDecimalRound(srcRate).divideDecimalRound(destRate);
    }

    function rateAndTimestampAtRound(bytes32 currencyKey, uint roundId) external view returns (uint rate, uint
time) {
        return _getRateAndTimestampAtRound(currencyKey, roundId);
    }

    function lastRateUpdateTimes(bytes32 currencyKey) external view returns (uint256) {
        return _getUpdatedTime(currencyKey);
```

```
    }

    function lastRateUpdateTimesForCurrencies(bytes32[] calldata currencyKeys) external view returns (uint[] memory) {
        uint[] memory lastUpdateTimes = new uint[](currencyKeys.length);

        for (uint i = 0; i < currencyKeys.length; i++) {
            lastUpdateTimes[i] = _getUpdatedTime(currencyKeys[i]);
        }

        return lastUpdateTimes;
    }

    function effectiveValue(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external view returns (uint value) {
        (value, , ) = _effectiveValueAndRates(sourceCurrencyKey, sourceAmount, destinationCurrencyKey);
    }

    function effectiveValueAndRates(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    )
        external
        view
        returns (
            uint value,
            uint sourceRate,
            uint destinationRate
        )
    {
        return _effectiveValueAndRates(sourceCurrencyKey, sourceAmount, destinationCurrencyKey);
    }

    function rateForCurrency(bytes32 currencyKey) external view returns (uint) {
        return _getRateAndUpdatedTime(currencyKey).rate;
    }

    function ratesAndUpdatedTimeForCurrencyLastNRounds(bytes32 currencyKey, uint numRounds)
        external
        view
        returns (uint[] memory rates, uint[] memory times)
    {
        rates = new uint[](numRounds);
        times = new uint[](numRounds);

        uint roundId = _getCurrentRoundId(currencyKey);
        for (uint i = 0; i < numRounds; i++) {
            // fetch the rate and treat is as current, so inverse limits if frozen will always be applied
            // regardless of current rate
            (rates[i], times[i]) = _getRateAndTimestampAtRound(currencyKey, roundId);

            if (roundId == 0) {
                // if we hit the last round, then return what we have
                return (rates, times);
            } else {
                roundId--;
            }
        }
    }

    function ratesForCurrencies(bytes32[] calldata currencyKeys) external view returns (uint[] memory) {
        uint[] memory _localRates = new uint[](currencyKeys.length);

        for (uint i = 0; i < currencyKeys.length; i++) {
            _localRates[i] = _getRate(currencyKeys[i]);
        }

        return _localRates;
    }

    function rateAndInvalid(bytes32 currencyKey) external view returns (uint rate, bool isInvalid) {
        RateAndUpdatedTime memory rateAndTime = _getRateAndUpdatedTime(currencyKey);

        if (currencyKey == "zUSD") {
            return (rateAndTime.rate, false);
        }
        return (
            rateAndTime.rate,
            _rateIsStaleWithTime(getRateStalePeriod(), rateAndTime.time) ||
                _rateIsFlagged(currencyKey, FlagsInterface(getAggregatorWarningFlags()))
        );
    }
```

```
function ratesAndInvalidForCurrencies(bytes32[] calldata currencyKeys)
    external
    view
    returns (uint[] memory rates, bool anyRateInvalid)
{
    rates = new uint[](currencyKeys.length);

    uint256 _rateStalePeriod = getRateStalePeriod();

    // fetch all flags at once
    bool[] memory flagList = getFlagsForRates(currencyKeys);

    for (uint i = 0; i < currencyKeys.length; i++) {
        // do one lookup of the rate & time to minimize gas
        RateAndUpdatedTime memory rateEntry = _getRateAndUpdatedTime(currencyKeys[i]);
        rates[i] = rateEntry.rate;
        if (!anyRateInvalid && currencyKeys[i] != "zUSD") {
            anyRateInvalid = flagList[i] || _rateIsStaleWithTime(_rateStalePeriod, rateEntry.time);
        }
    }
}

function rateIsStale(bytes32 currencyKey) external view returns (bool) {
    return _rateIsStale(currencyKey, getRateStalePeriod());
}

function rateIsFrozen(bytes32 currencyKey) external view returns (bool) {
    return _rateIsFrozen(currencyKey);
}

function rateIsInvalid(bytes32 currencyKey) external view returns (bool) {
    return
        _rateIsStale(currencyKey, getRateStalePeriod()) ||
        _rateIsFlagged(currencyKey, FlagsInterface(getAggregatorWarningFlags()));
}

function rateIsFlagged(bytes32 currencyKey) external view returns (bool) {
    return _rateIsFlagged(currencyKey, FlagsInterface(getAggregatorWarningFlags()));
}

function anyRateIsInvalid(bytes32[] calldata currencyKeys) external view returns (bool) {
    // Loop through each key and check whether the data point is stale.

    uint256 _rateStalePeriod = getRateStalePeriod();
    bool[] memory flagList = getFlagsForRates(currencyKeys);

    for (uint i = 0; i < currencyKeys.length; i++) {
        if (flagList[i] || _rateIsStale(currencyKeys[i], _rateStalePeriod)) {
            return true;
        }
    }

    return false;
}

/* ========== INTERNAL FUNCTIONS ========== */

function exchanger() internal view returns (IExchanger) {
    return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
}

function getFlagsForRates(bytes32[] memory currencyKeys) internal view returns (bool[] memory flagList) {
    // FlagsInterface _flags = FlagsInterface(getAggregatorWarningFlags());

    // // fetch all flags at once
    // if (_flags != FlagsInterface(0)) {
    //     address[] memory _aggregators = new address[](currencyKeys.length);

    //     for (uint i = 0; i < currencyKeys.length; i++) {
    //         _aggregators[i] = address(aggregators[currencyKeys[i]]);
    //     }

    //     flagList = _flags.getFlags(_aggregators);
    // } else {
    //     flagList = new bool[](currencyKeys.length);
    // }
    // set all currency flag to true
    flagList = new bool[](currencyKeys.length);//knownsec flagList 赋值为全 true,不再依赖 FlagsInterface
}

function _setRate(
    bytes32 currencyKey,
    uint256 rate,
    uint256 time
) internal {
    // Note: this will effectively start the rounds at 1, which matches Chainlink's Agggregators
    currentRoundForRate[currencyKey]++;
```

```
        _rates[currencyKey][currentRoundForRate[currencyKey]] = RateAndUpdatedTime({
            rate: uint216(rate),
            time: uint40(time)
        });
    }

    function internalUpdateRates(
        bytes32[] memory currencyKeys,
        uint[] memory newRates,
        uint timeSent
    ) internal returns (bool) {
        require(currencyKeys.length == newRates.length, "Currency key array length must match rates array length.");
        require(timeSent < (now + ORACLE_FUTURE_LIMIT), "Time is too far into the future");

        // Loop through each key and perform update.
        for (uint i = 0; i < currencyKeys.length; i++) {
            bytes32 currencyKey = currencyKeys[i];

            // Should not set any rate to zero ever, as no asset will ever be
            // truely worthless and still valid. In this scenario, we should
            // delete the rate and remove it from the system.
            require(newRates[i] != 0, "Zero is not a valid rate, please call deleteRate instead.");
            require(currencyKey != "zUSD", "Rate of zUSD cannot be updated, it's always UNIT.");

            // We should only update the rate if it's at least the same age as the last rate we've got.
            if (timeSent < _getUpdatedTime(currencyKey)) {
                continue;
            }

            // Ok, go ahead with the update.
            _setRate(currencyKey, newRates[i], timeSent);
        }

        emit RatesUpdated(currencyKeys, newRates);

        return true;
    }

    function removeFromArray(bytes32 entry, bytes32[] storage array) internal returns (bool) {
        for (uint i = 0; i < array.length; i++) {
            if (array[i] == entry) {
                delete array[i];

                // Copy the last key into the place of the one we just deleted
                // If there's only one key, this is array[0] = array[0].
                // If we're deleting the last one, it's also a NOOP in the same way.
                array[i] = array[array.length - 1];

                // Decrease the size of the array by one.
                array.length--;

                return true;
            }
        }
        return false;
    }

    function _rateOrInverted(
        bytes32 currencyKey,
        uint rate,
        uint roundId
    ) internal view returns (uint newRate) {
        // if an inverse mapping exists, adjust the price accordingly
        InversePricing memory inverse = inversePricing[currencyKey];
        if (inverse.entryPoint == 0 || rate == 0) {
            // when no inverse is set or when given a 0 rate, return the rate, regardless of the inverse status
            // (the latter is so when a new inverse is set but the underlying has no rate, it will return 0 as
            // the rate, not the lowerLimit)
            return rate;
        }

        newRate = rate;

        // Determine when round was frozen (if any)
        uint roundWhenRateFrozen = roundFrozen[currencyKey];
        // And if we're looking at a rate after frozen, and it's currently frozen, then apply the bounds limit even
        // if the current price is back within bounds
        if (roundId >= roundWhenRateFrozen && inverse.frozenAtUpperLimit) {
            newRate = inverse.upperLimit;
        } else if (roundId >= roundWhenRateFrozen && inverse.frozenAtLowerLimit) {
            newRate = inverse.lowerLimit;
        } else {
            // this ensures any rate outside the limit will never be returned
            uint doubleEntryPoint = inverse.entryPoint.mul(2);
            if (doubleEntryPoint <= rate) {
```

```
                // avoid negative numbers for unsigned ints, so set this to 0
                // which by the requirement that lowerLimit be > 0 will
                // cause this to freeze the price to the lowerLimit
                newRate = 0;
            } else {
                newRate = doubleEntryPoint.sub(rate);
            }

            // now ensure the rate is between the bounds
            if (newRate >= inverse.upperLimit) {
                newRate = inverse.upperLimit;
            } else if (newRate <= inverse.lowerLimit) {
                newRate = inverse.lowerLimit;
            }
        }
    }

    function _formatAggregatorAnswer(bytes32 currencyKey, uint256 rate) internal view returns (uint) {// knownsec 直接使用无符号整形接收
        require(rate >= 0, "Negative rate not supported");
        // if (currencyKeyDecimals[currencyKey] > 0) {
        //     uint multiplier = 10**uint(SafeMath.sub(18, 0));
        //     uint multiplier = 10**uint(SafeMath.sub(18, currencyKeyDecimals[currencyKey]));
        //     return uint(uint(rate).mul(multiplier));
        // }
        return uint(rate);// knownsec 直接返回转换后的rate
    }

    function bytes32ToString(bytes32 _bytes32, uint8 offset) public pure returns (string memory) {// knownsec bytes转string
        uint8 i = 0;
        while (i < 32 && _bytes32[i] != 0) {
            i++;
        }
        bytes memory bytesArray = new bytes(i - offset);
        for (i = 0 + offset; i < 32 && _bytes32[i] != 0; i++) {
            bytesArray[i - offset] = _bytes32[i];
        }
        return string(bytesArray);
    }

    function _getRateAndUpdatedTime(bytes32 currencyKey) internal view returns (RateAndUpdatedTime memory) {// knownsec 内部调用 构造rate和时间结构体
        // AggregatorV2V3Interface aggregator = aggregators[currencyKey];
        //
        // if (aggregator != AggregatorV2V3Interface(0)) {
        //     // this view from the aggregator is the most gas efficient but it can throw when there's no data,
        //     // so let's call it low-level to suppress any reverts
        //     bytes memory payload = abi.encodeWithSignature("latestRoundData()");
        //     // solhint-disable avoid-low-level-calls
        //     (bool success, bytes memory returnData) = address(aggregator).staticcall(payload);
        //
        //     if (success) {
        //         (uint80 roundId, int256 answer, , uint256 updatedAt, ) = abi.decode(
        //             returnData,
        //             (uint80, int256, uint256, uint256, uint80)
        //         );
        //         return
        //             RateAndUpdatedTime({
        //                                                 rate:    uint216(_rateOrInverted(currencyKey, _formatAggregatorAnswer(currencyKey, answer), roundId)),
        //                 time: uint40(updatedAt)
        //             });
        //     }
        // } else {

        // TODO change HZN Token's price feed for testnet
        if (bandProtocolOracle != IStdReference(0) && currencyKey != HZN) {// knownsec 预言机地址和标志检查
            // remove asset prefix
            uint8 offset = 1;
            // pass remove prefix for HZN currencyKey
            if (currencyKey == HZN) {
                currencyKey = SNX;
                offset = 0;
            }
            string memory stringCurrencyKey = bytes32ToString(currencyKey, offset);// knownsec 标志计算
            IStdReference.ReferenceData                memory                answer                =
bandProtocolOracle.getReferenceData(stringCurrencyKey, "USD");// knownsec 预言机信息交换
            uint256 updatedAt = answer.lastUpdatedBase >= answer.lastUpdatedQuote
                ? answer.lastUpdatedBase
                : answer.lastUpdatedQuote;
            uint roundId = currentRoundForRate[currencyKey];// knownsec 轮次获取
            return
                RateAndUpdatedTime({
                    rate:    uint216(_rateOrInverted(currencyKey,    _formatAggregatorAnswer(currencyKey, answer.rate), roundId)),
```

```
                time: uint40(updatedAt)
            });// knownsec 返回rate和时间结构体
        } else {
            uint roundId = currentRoundForRate[currencyKey];// knownsec 轮次获取
            RateAndUpdatedTime memory entry = _rates[currencyKey][roundId];

            return RateAndUpdatedTime({rate: uint216(_rateOrInverted(currencyKey, entry.rate, roundId)),
time: entry.time});// knownsec 返回rate和时间结构体
        }
    }

    function _getCurrentRoundId(bytes32 currencyKey) internal view returns (uint) {
        AggregatorV2V3Interface aggregator = aggregators[currencyKey];

        if (aggregator != AggregatorV2V3Interface(0)) {
            return aggregator.latestRound();
        } else {
            return currentRoundForRate[currencyKey];
        }
    }

    function _getRateAndTimestampAtRound(bytes32 currencyKey, uint roundId) internal view returns (uint rate,
uint time) {
        // AggregatorV2V3Interface aggregator = aggregators[currencyKey];

        // if (aggregator != AggregatorV2V3Interface(0)) {
        //     // this view from the aggregator is the most gas efficient but it can throw when there's no data,
        //     // so let's call it low-level to suppress any reverts
        //     bytes memory payload = abi.encodeWithSignature("getRoundData(uint80)", roundId);
        //     // solhint-disable avoid-low-level-calls
        //     (bool success, bytes memory returnData) = address(aggregator).staticcall(payload);

        //     if (success) {
        //         (, int256 answer, , uint256 updatedAt, ) = abi.decode(
        //             returnData,
        //             (uint80, int256, uint256, uint256, uint80)
        //         );
        //         return (_rateOrInverted(currencyKey, _formatAggregatorAnswer(currencyKey, answer),
roundId), updatedAt);
        //     }
        RateAndUpdatedTime memory update = _rates[currencyKey][roundId];// knownsec 使用_rates中
roundId
        return (_rateOrInverted(currencyKey, update.rate, roundId), update.time);
    }

    function _getRate(bytes32 currencyKey) internal view returns (uint256) {
        return _getRateAndUpdatedTime(currencyKey).rate;
    }

    function _getUpdatedTime(bytes32 currencyKey) internal view returns (uint256) {
        return _getRateAndUpdatedTime(currencyKey).time;
    }

    function _effectiveValueAndRates(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    )
        internal
        view
        returns (
            uint value,
            uint sourceRate,
            uint destinationRate
        )
    {
        sourceRate = _getRate(sourceCurrencyKey);
        // If there's no change in the currency, then just return the amount they gave us
        if (sourceCurrencyKey == destinationCurrencyKey) {
            destinationRate = sourceRate;
            value = sourceAmount;
        } else {
            // Calculate the effective value by going from source -> USD -> destination
            destinationRate = _getRate(destinationCurrencyKey);
            // prevent divide-by 0 error (this happens if the dest is not a valid rate)
            if (destinationRate > 0) {
                value                                                                                    =
sourceAmount.multiplyDecimalRound(sourceRate).divideDecimalRound(destinationRate);
            }
        }
    }

    function _rateIsStale(bytes32 currencyKey, uint _rateStalePeriod) internal view returns (bool) {
        // zUSD is a special case and is never stale (check before an SLOAD of getRateAndUpdatedTime)
        if (currencyKey == "zUSD") return false;

        return _rateIsStaleWithTime(_rateStalePeriod, _getUpdatedTime(currencyKey));
```

```
    }

    function _rateIsStaleWithTime(uint _rateStalePeriod, uint _time) internal view returns (bool) {
        return _time.add(_rateStalePeriod) < now;
    }

    function _rateIsFrozen(bytes32 currencyKey) internal view returns (bool) {
        InversePricing memory inverse = inversePricing[currencyKey];
        return inverse.frozenAtUpperLimit || inverse.frozenAtLowerLimit;
    }

    function _rateIsFlagged(bytes32 currencyKey, FlagsInterface flags) internal view returns (bool) {
        // zUSD is a special case and is never invalid
        if (currencyKey == "zUSD") return false;
        address aggregator = address(aggregators[currencyKey]);
        // when no aggregator or when the flags haven't been setup
        if (aggregator == address(0) || flags == FlagsInterface(0)) {
            return false;
        }
        return flags.getFlag(aggregator);
    }

    /* ========== MODIFIERS ========== */

    modifier onlyOracle {
        require(msg.sender == oracle, "Only the oracle can perform this action");
        _;
    }

    /* ========== EVENTS ========== */

    event OracleUpdated(address newOracle);
    event BandProtocolOracleUpdated(IStdReference newBandProtocolOracle);// knownsec 新增Band 预言机更新事件
    event RatesUpdated(bytes32[] currencyKeys, uint[] newRates);
    event RateDeleted(bytes32 currencyKey);
    event InversePriceConfigured(bytes32 currencyKey, uint entryPoint, uint upperLimit, uint lowerLimit);
    event InversePriceFrozen(bytes32 currencyKey, uint rate, uint roundId, address initiator);
    event AggregatorAdded(bytes32 currencyKey, address aggregator);
    event AggregatorRemoved(bytes32 currencyKey, address aggregator);
}
```

ExchangerWithVirtualSynth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Exchanger.sol";

// Internal references
import "./interfaces/IVirtualSynth.sol";
import "./VirtualSynth.sol";


// https://docs.synthetix.io/contracts/source/contracts/exchangerwithvirtualsynth
contract ExchangerWithVirtualSynth is Exchanger {
    constructor(address _owner, address _resolver) public Exchanger(_owner, _resolver) {}

    function _createVirtualSynth(
        IERC20 synth,
        address recipient,
        uint amount,
        bytes32 currencyKey
    ) internal returns (IVirtualSynth vSynth) {
        // prevent inverse zassets from being allowed due to purgeability
        // toAscii(0x69) -> i
        require(currencyKey[0] != 0x69, "Cannot virtualize this zasset");

        vSynth = new VirtualSynth(synth, resolver, recipient, amount, currencyKey);
        emit VirtualSynthCreated(address(synth), recipient, address(vSynth), currencyKey, amount);
    }

    event VirtualSynthCreated(
        address indexed synth,
        address indexed recipient,
        address vSynth,
        bytes32 currencyKey,
        uint amount
    );
}
```

ExchangeState.sol

```
pragma solidity ^0.5.16;
```

```
// Inheritance
import "./Owned.sol";
import "./State.sol";
import "./interfaces/IExchangeState.sol";


// https://docs.synthetix.io/contracts/source/contracts/exchangestate
contract ExchangeState is Owned, State, IExchangeState {
    mapping(address => mapping(bytes32 => IExchangeState.ExchangeEntry[])) public exchanges;

    uint public maxEntriesInQueue = 0;// knownsec 初始化不允许用户实体、后续 owner 可用 setMaxEntriesInQueue 修改

    constructor(address _owner, address _associatedContract) public Owned(_owner) State(_associatedContract) {}

    /* ========== SETTERS ========== */

    function setMaxEntriesInQueue(uint _maxEntriesInQueue) external onlyOwner {
        maxEntriesInQueue = _maxEntriesInQueue;
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function appendExchangeEntry(
        address account,
        bytes32 src,
        uint amount,
        bytes32 dest,
        uint amountReceived,
        uint exchangeFeeRate,
        uint timestamp,
        uint roundIdForSrc,
        uint roundIdForDest
    ) external onlyAssociatedContract {
        require(exchanges[account][dest].length < maxEntriesInQueue, "Max queue length reached");

        exchanges[account][dest].push(
            ExchangeEntry({
                src: src,
                amount: amount,
                dest: dest,
                amountReceived: amountReceived,
                exchangeFeeRate: exchangeFeeRate,
                timestamp: timestamp,
                roundIdForSrc: roundIdForSrc,
                roundIdForDest: roundIdForDest
            })
        );
    }

    function removeEntries(address account, bytes32 currencyKey) external onlyAssociatedContract {
        delete exchanges[account][currencyKey];
    }

    /* ========== VIEWS ========== */

    function getLengthOfEntries(address account, bytes32 currencyKey) external view returns (uint) {
        return exchanges[account][currencyKey].length;
    }

    function getEntryAt(
        address account,
        bytes32 currencyKey,
        uint index
    )
        external
        view
        returns (
            bytes32 src,
            uint amount,
            bytes32 dest,
            uint amountReceived,
            uint exchangeFeeRate,
            uint timestamp,
            uint roundIdForSrc,
            uint roundIdForDest
        )
    {
        ExchangeEntry storage entry = exchanges[account][currencyKey][index];
        return (
            entry.src,
            entry.amount,
            entry.dest,
            entry.amountReceived,
            entry.exchangeFeeRate,
            entry.timestamp,
```

```
                entry.roundIdForSrc,
                entry.roundIdForDest
            );
        }

        function getMaxTimestamp(address account, bytes32 currencyKey) external view returns (uint) {
            ExchangeEntry[] storage userEntries = exchanges[account][currencyKey];
            uint timestamp = 0;
            for (uint i = 0; i < userEntries.length; i++) {
                if (userEntries[i].timestamp > timestamp) {
                    timestamp = userEntries[i].timestamp;
                }
            }
            return timestamp;
        }
}
```

ExternStateToken.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Proxyable.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./TokenState.sol";


// https://docs.synthetix.io/contracts/source/contracts/externstatetoken
contract ExternStateToken is Owned, Proxyable {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    /* ========== STATE VARIABLES ========== */

    /* Stores balances and allowances. */
    TokenState public tokenState;

    /* Other ERC20 fields. */
    string public name;
    string public symbol;
    uint public totalSupply;
    uint8 public decimals;

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        string memory _name,
        string memory _symbol,
        uint _totalSupply,
        uint8 _decimals,
        address _owner
    ) public Owned(_owner) Proxyable(_proxy) {
        tokenState = _tokenState;

        name = _name;
        symbol = _symbol;
        totalSupply = _totalSupply;
        decimals = _decimals;
    }

    /* ========== VIEWS ========== */

    /**
      * @notice Returns the ERC20 allowance of one party to spend on behalf of another.
      * @param owner The party authorising spending of their funds.
      * @param spender The party spending tokenOwner's funds.
      */
    function allowance(address owner, address spender) public view returns (uint) {
        return tokenState.allowance(owner, spender);
    }

    /**
      * @notice Returns the ERC20 token balance of a given account.
      */
    function balanceOf(address account) external view returns (uint) {
        return tokenState.balanceOf(account);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /**
```

```
     * @notice Set the address of the TokenState contract.
     * @dev This can be used to "pause" transfer functionality, by pointing the tokenState at 0x000..
     * as balances would be unreachable.
     */
    function setTokenState(TokenState _tokenState) external optionalProxy_onlyOwner {
        tokenState = _tokenState;
        emitTokenStateUpdated(address(_tokenState));
    }

    function _internalTransfer(
        address from,
        address to,
        uint value
    ) internal returns (bool) {
        /* Disallow transfers to irretrievable-addresses. */
        require(to != address(0) && to != address(this) && to != address(proxy), "Cannot transfer to this address");

        // Insufficient balance will be handled by the safe subtraction.
        tokenState.setBalanceOf(from, tokenState.balanceOf(from).sub(value));
        tokenState.setBalanceOf(to, tokenState.balanceOf(to).add(value));

        // Emit a standard ERC20 transfer event
        emitTransfer(from, to, value);

        return true;
    }

    /**
     * @dev Perform an ERC20 token transfer. Designed to be called by transfer functions possessing
     * the onlyProxy or optionalProxy modifiers.
     */
    function _transferByProxy(
        address from,
        address to,
        uint value
    ) internal returns (bool) {
        return _internalTransfer(from, to, value);
    }

    /*
     * @dev Perform an ERC20 token transferFrom. Designed to be called by transferFrom functions
     * possessing the optionalProxy or optionalProxy modifiers.
     */
    function _transferFromByProxy(
        address sender,
        address from,
        address to,
        uint value
    ) internal returns (bool) {
        /* Insufficient allowance will be handled by the safe subtraction. */
        tokenState.setAllowance(from, sender, tokenState.allowance(from, sender).sub(value));
        return _internalTransfer(from, to, value);
    }

    /**
     * @notice Approves spender to transfer on the message sender's behalf.
     */
    function approve(address spender, uint value) public optionalProxy returns (bool) {
        address sender = messageSender;

        tokenState.setAllowance(sender, spender, value);
        emitApproval(sender, spender, value);
        return true;
    }

    /* ========== EVENTS ========== */
    function addressToBytes32(address input) internal pure returns (bytes32) {
        return bytes32(uint256(uint160(input)));
    }

    event Transfer(address indexed from, address indexed to, uint value);
    bytes32 internal constant TRANSFER_SIG = keccak256("Transfer(address,address,uint256)");

    function emitTransfer(
        address from,
        address to,
        uint value
    ) internal {
        proxy._emit(abi.encode(value), 3, TRANSFER_SIG, addressToBytes32(from), addressToBytes32(to), 0);
    }

    event Approval(address indexed owner, address indexed spender, uint value);
    bytes32 internal constant APPROVAL_SIG = keccak256("Approval(address,address,uint256)");

    function emitApproval(
        address owner,
```

```
        address spender,
        uint value
    ) internal {
        proxy._emit(abi.encode(value), 3, APPROVAL_SIG, addressToBytes32(owner),
addressToBytes32(spender), 0);
    }

    event TokenStateUpdated(address newTokenState);
    bytes32 internal constant TOKENSTATEUPDATED_SIG = keccak256("TokenStateUpdated(address)");

    function emitTokenStateUpdated(address newTokenState) internal {
        proxy._emit(abi.encode(newTokenState), 1, TOKENSTATEUPDATED_SIG, 0, 0, 0);
    }
}
```

FeePool.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./Proxyable.sol";
import "./LimitedSetup.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/IFeePool.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IERC20.sol";
import "./interfaces/ISynth.sol";
import "./interfaces/ISystemStatus.sol";
import "./interfaces/ISynthetix.sol";
import "./FeePoolState.sol";
import "./FeePoolEternalStorage.sol";
import "./interfaces/IExchanger.sol";
import "./interfaces/IIssuer.sol";
import "./interfaces/ISynthetixState.sol";
import "./interfaces/IRewardEscrow.sol";
import "./interfaces/IDelegateApprovals.sol";
import "./interfaces/IRewardsDistribution.sol";
import "./interfaces/IEtherCollateralsUSD.sol";


// https://docs.synthetix.io/contracts/source/contracts/feepool
contract FeePool is Owned, Proxyable, LimitedSetup, MixinResolver, MixinSystemSettings, IFeePool {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    // Where fees are pooled in zUSD.
    address public constant FEE_ADDRESS = 0xfeEFEEfeefEeFeefEEFEEfEeFeefEEFeeFEEFEeF;

    // zUSD currencyKey. Fees stored and paid in zUSD
    bytes32 private zUSD = "zUSD";

    // This struct represents the issuance activity that's happened in a fee period.
    struct FeePeriod {
        uint64 feePeriodId;
        uint64 startingDebtIndex;
        uint64 startTime;
        uint feesToDistribute;
        uint feesClaimed;
        uint rewardsToDistribute;
        uint rewardsClaimed;
    }

    // A staker(mintr) can claim from the previous fee period (7 days) only.
    // Fee Periods stored and managed from [0], such that [0] is always
    // the current active fee period which is not claimable until the
    // public function closeCurrentFeePeriod() is called closing the
    // current weeks collected fees. [1] is last weeks feeperiod
    uint8 public constant FEE_PERIOD_LENGTH = 2;

    FeePeriod[FEE_PERIOD_LENGTH] private _recentFeePeriods;
    uint256 private _currentFeePeriod;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_FEEPOOLSTATE = "FeePoolState";
    bytes32 private constant CONTRACT_FEEPOOLETERNALSTORAGE = "FeePoolEternalStorage";
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
```

```
    bytes32 private constant CONTRACT_SYNTHETIXSTATE = "SynthetixState";
    bytes32 private constant CONTRACT_REWARDESCROW = "RewardEscrow";
    bytes32 private constant CONTRACT_DELEGATEAPPROVALS = "DelegateApprovals";
    bytes32 private constant CONTRACT_ETH_COLLATERAL_SUSD = "EtherCollateralsUSD";
    bytes32 private constant CONTRACT_REWARDSDISTRIBUTION = "RewardsDistribution";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_SYNTHETIX,
        CONTRACT_FEEPOOLSTATE,
        CONTRACT_FEEPOOLETERNALSTORAGE,
        CONTRACT_EXCHANGER,
        CONTRACT_ISSUER,
        CONTRACT_SYNTHETIXSTATE,
        CONTRACT_REWARDESCROW,
        CONTRACT_DELEGATEAPPROVALS,
        CONTRACT_ETH_COLLATERAL_SUSD,
        CONTRACT_REWARDSDISTRIBUTION
    ];

    /* ========== ETERNAL STORAGE CONSTANTS ========== */

    bytes32 private constant LAST_FEE_WITHDRAWAL = "last_fee_withdrawal";

    constructor(
        address payable _proxy,
        address _owner,
        address _resolver
    )
        public
        Owned(_owner)
        Proxyable(_proxy)
        LimitedSetup(3 weeks)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {
        // Set our initial fee period
        _recentFeePeriodsStorage(0).feePeriodId = 1;
        _recentFeePeriodsStorage(0).startTime = uint64(now);
    }

    /* ========== VIEWS ========== */

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function feePoolState() internal view returns (FeePoolState) {
        return FeePoolState(requireAndGetAddress(CONTRACT_FEEPOOLSTATE, "Missing FeePoolState address"));
    }

    function feePoolEternalStorage() internal view returns (FeePoolEternalStorage) {
        return
            FeePoolEternalStorage(
                requireAndGetAddress(CONTRACT_FEEPOOLETERNALSTORAGE, "Missing FeePoolEternalStorage address")
            );
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function etherCollateralsUSD() internal view returns (IEtherCollateralsUSD) {
        return
            IEtherCollateralsUSD(requireAndGetAddress(CONTRACT_ETH_COLLATERAL_SUSD, "Missing EtherCollateralsUSD address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function synthetixState() internal view returns (ISynthetixState) {
        return ISynthetixState(requireAndGetAddress(CONTRACT_SYNTHETIXSTATE, "Missing HorizonState address"));
    }

    function rewardEscrow() internal view returns (IRewardEscrow) {
        return IRewardEscrow(requireAndGetAddress(CONTRACT_REWARDESCROW, "Missing RewardEscrow address"));
```

```
    }

    function delegateApprovals() internal view returns (IDelegateApprovals) {
        return IDelegateApprovals(requireAndGetAddress(CONTRACT_DELEGATEAPPROVALS, "Missing
DelegateApprovals address"));
    }

    function rewardsDistribution() internal view returns (IRewardsDistribution) {
        return
            IRewardsDistribution(requireAndGetAddress(CONTRACT_REWARDSDISTRIBUTION, "Missing
RewardsDistribution address"));
    }

    function issuanceRatio() external view returns (uint) {
        return getIssuanceRatio();
    }

    function feePeriodDuration() external view returns (uint) {
        return getFeePeriodDuration();
    }

    function targetThreshold() external view returns (uint) {
        return getTargetThreshold();
    }

    function recentFeePeriods(uint index)
        external
        view
        returns (
            uint64 feePeriodId,
            uint64 startingDebtIndex,
            uint64 startTime,
            uint feesToDistribute,
            uint feesClaimed,
            uint rewardsToDistribute,
            uint rewardsClaimed
        )
    {
        FeePeriod memory feePeriod = _recentFeePeriodsStorage(index);
        return (
            feePeriod.feePeriodId,
            feePeriod.startingDebtIndex,
            feePeriod.startTime,
            feePeriod.feesToDistribute,
            feePeriod.feesClaimed,
            feePeriod.rewardsToDistribute,
            feePeriod.rewardsClaimed
        );
    }

    function _recentFeePeriodsStorage(uint index) internal view returns (FeePeriod storage) {
        return _recentFeePeriods[(_currentFeePeriod + index) % FEE_PERIOD_LENGTH];
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /**
     * @notice Logs an accounts issuance data per fee period
     * @param account Message.Senders account address
     * @param debtRatio Debt percentage this account has locked after minting or burning their zasset
     * @param debtEntryIndex The index in the global debt ledger. synthetixState.issuanceData(account)
     * @dev onlyIssuer to call me on synthetix.issue() & synthetix.burn() calls to store the locked HZN
     * per fee period so we know to allocate the correct proportions of fees and rewards per period
     */
    function appendAccountIssuanceRecord(
        address account,
        uint debtRatio,
        uint debtEntryIndex
    ) external onlyIssuer {
        feePoolState().appendAccountIssuanceRecord(
            account,
            debtRatio,
            debtEntryIndex,
            _recentFeePeriodsStorage(0).startingDebtIndex
        );

        emitIssuanceDebtRatioEntry(account, debtRatio, debtEntryIndex,
_recentFeePeriodsStorage(0).startingDebtIndex);
    }

    /**
     * @notice The Exchanger contract informs us when fees are paid.
     * @param amount zUSD amount in fees being paid.
     */
    function recordFeePaid(uint amount) external onlyInternalContracts {
        // Keep track off fees in zUSD in the open fee pool period.
        _recentFeePeriodsStorage(0).feesToDistribute =
```

```
_recentFeePeriodsStorage(0).feesToDistribute.add(amount);
    }

    /**
     * @notice The RewardsDistribution contract informs us how many HZN rewards are sent to RewardEscrow to be claimed.
     */
    function setRewardsToDistribute(uint amount) external {
        address rewardsAuthority = address(rewardsDistribution());
        require(messageSender == rewardsAuthority || msg.sender == rewardsAuthority, "Caller is not rewardsAuthority");
        // Add the amount of HZN rewards to distribute on top of any rolling unclaimed amount
        _recentFeePeriodsStorage(0).rewardsToDistribute = _recentFeePeriodsStorage(0).rewardsToDistribute.add(amount);
    }

    /**
     * @notice Close the current fee period and start a new one.
     */
    function closeCurrentFeePeriod() external issuanceActive {
        require(getFeePeriodDuration() > 0, "Fee Period Duration not set");
        require(_recentFeePeriodsStorage(0).startTime <= (now - getFeePeriodDuration()), "Too early to close fee period");

        // Note:    when FEE_PERIOD_LENGTH = 2, periodClosing is the current period & periodToRollover is the last open claimable period
        FeePeriod storage periodClosing = _recentFeePeriodsStorage(FEE_PERIOD_LENGTH - 2);
        FeePeriod storage periodToRollover = _recentFeePeriodsStorage(FEE_PERIOD_LENGTH - 1);

        // Any unclaimed fees from the last period in the array roll back one period.
        // Because of the subtraction here, they're effectively proportionally redistributed to those who
        // have already claimed from the old period, available in the new period.
        // The subtraction is important so we don't create a ticking time bomb of an ever growing
        // number of fees that can never decrease and will eventually overflow at the end of the fee pool.
        _recentFeePeriodsStorage(FEE_PERIOD_LENGTH - 2).feesToDistribute = periodToRollover
            .feesToDistribute
            .sub(periodToRollover.feesClaimed)
            .add(periodClosing.feesToDistribute);
        _recentFeePeriodsStorage(FEE_PERIOD_LENGTH - 2).rewardsToDistribute = periodToRollover
            .rewardsToDistribute
            .sub(periodToRollover.rewardsClaimed)
            .add(periodClosing.rewardsToDistribute);

        // Shift the previous fee periods across to make room for the new one.
        _currentFeePeriod = _currentFeePeriod.add(FEE_PERIOD_LENGTH).sub(1).mod(FEE_PERIOD_LENGTH);

        // Clear the first element of the array to make sure we don't have any stale values.
        delete _recentFeePeriods[_currentFeePeriod];

        // Open up the new fee period.
        // Increment periodId from the recent closed period feePeriodId
        _recentFeePeriodsStorage(0).feePeriodId = uint64(uint256(_recentFeePeriodsStorage(1).feePeriodId).add(1));
        _recentFeePeriodsStorage(0).startingDebtIndex = uint64(synthetixState().debtLedgerLength());
        _recentFeePeriodsStorage(0).startTime = uint64(now);

        emitFeePeriodClosed(_recentFeePeriodsStorage(1).feePeriodId);
    }

    /**
     * @notice Claim fees for last period when available or not already withdrawn.
     */
    function claimFees() external issuanceActive optionalProxy returns (bool) {
        return _claimFees(messageSender);
    }

    /**
     * @notice Delegated claimFees(). Call from the deletegated address
     * and the fees will be sent to the claimingForAddress.
     * approveClaimOnBehalf() must be called first to approve the deletage address
     * @param claimingForAddress The account you are claiming fees for
     */
    function claimOnBehalf(address claimingForAddress) external issuanceActive optionalProxy returns (bool) {
        require(delegateApprovals().canClaimFor(claimingForAddress, messageSender), "Not approved to claim on behalf");

        return _claimFees(claimingForAddress);
    }

    function _claimFees(address claimingAddress) internal returns (bool) {
        uint rewardsPaid = 0;
        uint feesPaid = 0;
        uint availableFees;
        uint availableRewards;

        // Address won't be able to claim fees if it is too far below the target c-ratio.
```

```
        // It will need to burn synths then try claiming again.
        (bool feesClaimable, bool anyRateIsInvalid) = _isFeesClaimableAndAnyRatesInvalid(claimingAddress);

        require(feesClaimable, "C-Ratio below penalty threshold");

        require(!anyRateIsInvalid, "A zasset or HZN rate is invalid");

        // Get the claimingAddress available fees and rewards
        (availableFees, availableRewards) = feesAvailable(claimingAddress);

        require(
            availableFees > 0 || availableRewards > 0,
            "No fees or rewards available for period, or fees already claimed"
        );

        // Record the address has claimed for this period
        _setLastFeeWithdrawal(claimingAddress, _recentFeePeriodsStorage(1).feePeriodId);

        if (availableFees > 0) {
            // Record the fee payment in our recentFeePeriods
            feesPaid = _recordFeePayment(availableFees);

            // Send them their fees
            _payFees(claimingAddress, feesPaid);
        }

        if (availableRewards > 0) {
            // Record the reward payment in our recentFeePeriods
            rewardsPaid = _recordRewardPayment(availableRewards);

            // Send them their rewards
            _payRewards(claimingAddress, rewardsPaid);
        }

        emitFeesClaimed(claimingAddress, feesPaid, rewardsPaid);

        return true;
    }

    /**
     * @notice Admin function to import the FeePeriod data from the previous contract
     */
    function importFeePeriod(
        uint feePeriodIndex,
        uint feePeriodId,
        uint startingDebtIndex,
        uint startTime,
        uint feesToDistribute,
        uint feesClaimed,
        uint rewardsToDistribute,
        uint rewardsClaimed
    ) public optionalProxy_onlyOwner onlyDuringSetup {
        require(startingDebtIndex <= synthetixState().debtLedgerLength(), "Cannot import bad data");

        _recentFeePeriods[_currentFeePeriod.add(feePeriodIndex).mod(FEE_PERIOD_LENGTH)] =
FeePeriod({
            feePeriodId: uint64(feePeriodId),
            startingDebtIndex: uint64(startingDebtIndex),
            startTime: uint64(startTime),
            feesToDistribute: feesToDistribute,
            feesClaimed: feesClaimed,
            rewardsToDistribute: rewardsToDistribute,
            rewardsClaimed: rewardsClaimed
        });
    }

    /**
     * @notice Owner can escrow HZN. Owner to send the tokens to the RewardEscrow
     * @param account Address to escrow tokens for
     * @param quantity Amount of tokens to escrow
     */
    function appendVestingEntry(address account, uint quantity) public optionalProxy_onlyOwner {
        // Transfer HZN from messageSender to the Reward Escrow
        IERC20(address(synthetix())).transferFrom(messageSender, address(rewardEscrow()), quantity);

        // Create Vesting Entry
        rewardEscrow().appendVestingEntry(account, quantity);
    }

    /**
     * @notice Record the fee payment in our recentFeePeriods.
     * @param zUSDAmount The amount of fees priced in zUSD.
     */
    function _recordFeePayment(uint zUSDAmount) internal returns (uint) {
        // Don't assign to the parameter
        uint remainingToAllocate = zUSDAmount;
```

```
        uint feesPaid;
        // Start at the oldest period and record the amount, moving to newer periods
        // until we've exhausted the amount.
        // The condition checks for overflow because we're going to 0 with an unsigned int.
        for (uint i = FEE_PERIOD_LENGTH - 1; i < FEE_PERIOD_LENGTH; i--) {
            uint feesAlreadyClaimed = _recentFeePeriodsStorage(i).feesClaimed;
            uint delta = _recentFeePeriodsStorage(i).feesToDistribute.sub(feesAlreadyClaimed);

            if (delta > 0) {
                // Take the smaller of the amount left to claim in the period and the amount we need to allocate
                uint amountInPeriod = delta < remainingToAllocate ? delta : remainingToAllocate;

                _recentFeePeriodsStorage(i).feesClaimed = feesAlreadyClaimed.add(amountInPeriod);
                remainingToAllocate = remainingToAllocate.sub(amountInPeriod);
                feesPaid = feesPaid.add(amountInPeriod);

                // No need to continue iterating if we've recorded the whole amount;
                if (remainingToAllocate == 0) return feesPaid;

                // We've exhausted feePeriods to distribute and no fees remain in last period
                // User last to claim would in this scenario have their remainder slashed
                if (i == 0 && remainingToAllocate > 0) {
                    remainingToAllocate = 0;
                }
            }
        }

        return feesPaid;
    }

    /**
     * @notice Record the reward payment in our recentFeePeriods.
     * @param hznAmount The amount of HZN tokens.
     */
    function _recordRewardPayment(uint hznAmount) internal returns (uint) {
        // Don't assign to the parameter
        uint remainingToAllocate = hznAmount;

        uint rewardPaid;

        // Start at the oldest period and record the amount, moving to newer periods
        // until we've exhausted the amount.
        // The condition checks for overflow because we're going to 0 with an unsigned int.
        for (uint i = FEE_PERIOD_LENGTH - 1; i < FEE_PERIOD_LENGTH; i--) {
            uint toDistribute = _recentFeePeriodsStorage(i).rewardsToDistribute.sub(
                _recentFeePeriodsStorage(i).rewardsClaimed
            );

            if (toDistribute > 0) {
                // Take the smaller of the amount left to claim in the period and the amount we need to allocate
                uint amountInPeriod = toDistribute < remainingToAllocate ? toDistribute : remainingToAllocate;

                _recentFeePeriodsStorage(i).rewardsClaimed = _recentFeePeriodsStorage(i).rewardsClaimed.add(amountInPeriod);
                remainingToAllocate = remainingToAllocate.sub(amountInPeriod);
                rewardPaid = rewardPaid.add(amountInPeriod);

                // No need to continue iterating if we've recorded the whole amount;
                if (remainingToAllocate == 0) return rewardPaid;

                // We've exhausted feePeriods to distribute and no rewards remain in last period
                // User last to claim would in this scenario have their remainder slashed
                // due to rounding up of PreciseDecimal
                if (i == 0 && remainingToAllocate > 0) {
                    remainingToAllocate = 0;
                }
            }
        }
        return rewardPaid;
    }

    /**
     * @notice Send the fees to claiming address.
     * @param account The address to send the fees to.
     * @param zUSDAmount The amount of fees priced in zUSD.
     */
    function _payFees(address account, uint zUSDAmount) internal notFeeAddress(account) {
        // Grab the zUSD Synth
        ISynth zUSDSynth = issuer().synths(zUSD);

        // NOTE: we do not control the FEE_ADDRESS so it is not possible to do an
        // ERC20.approve() transaction to allow this feePool to call ERC20.transferFrom
        // to the accounts address

        // Burn the source amount
        zUSDSynth.burn(FEE_ADDRESS, zUSDAmount);
```

```
        // Mint their new synths
        zUSDSynth.issue(account, zUSDAmount);
    }

    /**
     * @notice Send the rewards to claiming address - will be locked in rewardEscrow.
     * @param account The address to send the fees to.
     * @param hznAmount The amount of HZN.
     */
    function _payRewards(address account, uint hznAmount) internal notFeeAddress(account) {
        // Record vesting entry for claiming address and amount
        // HZN already minted to rewardEscrow balance
        rewardEscrow().appendVestingEntry(account, hznAmount);
    }

    /**
     * @notice The total fees available in the system to be withdrawn in zUSD
     */
    function totalFeesAvailable() external view returns (uint) {
        uint totalFees = 0;

        // Fees in fee period [0] are not yet available for withdrawal
        for (uint i = 1; i < FEE_PERIOD_LENGTH; i++) {
            totalFees = totalFees.add(_recentFeePeriodsStorage(i).feesToDistribute);
            totalFees = totalFees.sub(_recentFeePeriodsStorage(i).feesClaimed);
        }

        return totalFees;
    }

    /**
     * @notice The total HZN rewards available in the system to be withdrawn
     */
    function totalRewardsAvailable() external view returns (uint) {
        uint totalRewards = 0;

        // Rewards in fee period [0] are not yet available for withdrawal
        for (uint i = 1; i < FEE_PERIOD_LENGTH; i++) {
            totalRewards = totalRewards.add(_recentFeePeriodsStorage(i).rewardsToDistribute);
            totalRewards = totalRewards.sub(_recentFeePeriodsStorage(i).rewardsClaimed);
        }

        return totalRewards;
    }

    /**
     * @notice The fees available to be withdrawn by a specific account, priced in zUSD
     * @dev Returns two amounts, one for fees and one for HZN rewards
     */
    function feesAvailable(address account) public view returns (uint, uint) {
        // Add up the fees
        uint[2][FEE_PERIOD_LENGTH] memory userFees = feesByPeriod(account);

        uint totalFees = 0;
        uint totalRewards = 0;

        // Fees & Rewards in fee period [0] are not yet available for withdrawal
        for (uint i = 1; i < FEE_PERIOD_LENGTH; i++) {
            totalFees = totalFees.add(userFees[i][0]);
            totalRewards = totalRewards.add(userFees[i][1]);
        }

        // And convert totalFees to zUSD
        // Return totalRewards as is in HZN amount
        return (totalFees, totalRewards);
    }

    function _isFeesClaimableAndAnyRatesInvalid(address account) internal view returns (bool, bool) {
        // Threshold is calculated from ratio % above the target ratio (issuanceRatio).
        //    0    <    10%:      Claimable
        // 10% > above:    Unable to claim
        (uint ratio, bool anyRateIsInvalid) = issuer().collateralisationRatioAndAnyRatesInvalid(account);
        uint targetRatio = getIssuanceRatio();

        // Claimable if collateral ratio below target ratio
        if (ratio < targetRatio) {
            return (true, anyRateIsInvalid);
        }

        // Calculate the threshold for collateral ratio before fees can't be claimed.
        uint ratio_threshold = targetRatio.multiplyDecimal(SafeDecimalMath.unit().add(getTargetThreshold()));

        // Not claimable if collateral ratio above threshold
        if (ratio > ratio_threshold) {
            return (false, anyRateIsInvalid);
        }
```

```
        return (true, anyRateIsInvalid);
    }

    function isFeesClaimable(address account) external view returns (bool feesClaimable) {
        (feesClaimable, ) = _isFeesClaimableAndAnyRatesInvalid(account);
    }

    /**
     * @notice Calculates fees by period for an account, priced in zUSD
     * @param account The address you want to query the fees for
     */
    function feesByPeriod(address account) public view returns (uint[2][FEE_PERIOD_LENGTH] memory results) {
        // What's the user's debt entry index and the debt they owe to the system at current feePeriod
        uint userOwnershipPercentage;
        uint debtEntryIndex;
        FeePoolState _feePoolState = feePoolState();

        (userOwnershipPercentage, debtEntryIndex) = _feePoolState.getAccountsDebtEntry(account, 0);

        // If they don't have any debt ownership and they never minted, they don't have any fees.
        // User ownership can reduce to 0 if user burns all synths,
        // however they could have fees applicable for periods they had minted in before so we check debtEntryIndex.
        if (debtEntryIndex == 0 && userOwnershipPercentage == 0) {
            uint[2][FEE_PERIOD_LENGTH] memory nullResults;
            return nullResults;
        }

        // The [0] fee period is not yet ready to claim, but it is a fee period that they can have
        // fees owing for, so we need to report on it anyway.
        uint feesFromPeriod;
        uint rewardsFromPeriod;
        (feesFromPeriod, rewardsFromPeriod) = _feesAndRewardsFromPeriod(0, userOwnershipPercentage, debtEntryIndex);

        results[0][0] = feesFromPeriod;
        results[0][1] = rewardsFromPeriod;

        // Retrieve user's last fee claim by periodId
        uint lastFeeWithdrawal = getLastFeeWithdrawal(account);

        // Go through our fee periods from the oldest feePeriod[FEE_PERIOD_LENGTH - 1] and figure out what we owe them.
        // Condition checks for periods > 0
        for (uint i = FEE_PERIOD_LENGTH - 1; i > 0; i--) {
            uint next = i - 1;
            uint nextPeriodStartingDebtIndex = _recentFeePeriodsStorage(next).startingDebtIndex;

            // We can skip the period, as no debt minted during period (next period's startingDebtIndex is still 0)
            if (nextPeriodStartingDebtIndex > 0 && lastFeeWithdrawal < _recentFeePeriodsStorage(i).feePeriodId) {
                // We calculate a feePeriod's closingDebtIndex by looking at the next feePeriod's startingDebtIndex
                // we can use the most recent issuanceData[0] for the current feePeriod
                // else find the applicableIssuanceData for the feePeriod based on the StartingDebtIndex of the period
                uint closingDebtIndex = uint256(nextPeriodStartingDebtIndex).sub(1);

                // Gas optimisation - to reuse debtEntryIndex if found new applicable one
                // if applicable is 0,0 (none found) we keep most recent one from issuanceData[0]
                // return if userOwnershipPercentage = 0)
                (userOwnershipPercentage, debtEntryIndex) = _feePoolState.applicableIssuanceData(account, closingDebtIndex);

                (feesFromPeriod, rewardsFromPeriod) = _feesAndRewardsFromPeriod(i, userOwnershipPercentage, debtEntryIndex);

                results[i][0] = feesFromPeriod;
                results[i][1] = rewardsFromPeriod;
            }
        }
    }

    /**
     * @notice ownershipPercentage is a high precision decimals uint based on
     * wallet's debtPercentage. Gives a precise amount of the feesToDistribute
     * for fees in the period. Precision factor is removed before results are
     * returned.
     * @dev The reported fees owing for the current period [0] are just a
     * running balance until the fee period closes
     */
    function _feesAndRewardsFromPeriod(
        uint period,
        uint ownershipPercentage,
        uint debtEntryIndex
```

```
    ) internal view returns (uint, uint) {
        // If it's zero, they haven't issued, and they have no fees OR rewards.
        if (ownershipPercentage == 0) return (0, 0);

        uint debtOwnershipForPeriod = ownershipPercentage;

        // If period has closed we want to calculate debtPercentage for the period
        if (period > 0) {
            uint closingDebtIndex = uint256(_recentFeePeriodsStorage(period - 1).startingDebtIndex).sub(1);
            debtOwnershipForPeriod = _effectiveDebtRatioForPeriod(closingDebtIndex,
ownershipPercentage, debtEntryIndex);
        }

        // Calculate their percentage of the fees / rewards in this period
        // This is a high precision integer.
        uint feesFromPeriod =
_recentFeePeriodsStorage(period).feesToDistribute.multiplyDecimal(debtOwnershipForPeriod);

        uint rewardsFromPeriod = _recentFeePeriodsStorage(period).rewardsToDistribute.multiplyDecimal(
            debtOwnershipForPeriod
        );

        return (feesFromPeriod.preciseDecimalToDecimal(), rewardsFromPeriod.preciseDecimalToDecimal());
    }

    function _effectiveDebtRatioForPeriod(
        uint closingDebtIndex,
        uint ownershipPercentage,
        uint debtEntryIndex
    ) internal view returns (uint) {
        // Figure out their global debt percentage delta at end of fee Period.
        // This is a high precision integer.
        ISynthetixState _synthetixState = synthetixState();
        uint feePeriodDebtOwnership = _synthetixState
            .debtLedger(closingDebtIndex)
            .divideDecimalRoundPrecise(_synthetixState.debtLedger(debtEntryIndex))
            .multiplyDecimalRoundPrecise(ownershipPercentage);

        return feePeriodDebtOwnership;
    }

    function effectiveDebtRatioForPeriod(address account, uint period) external view returns (uint) {
        require(period != 0, "Current period is not closed yet");
        require(period < FEE_PERIOD_LENGTH, "Exceeds the FEE_PERIOD_LENGTH");

        // If the period being checked is uninitialised then return 0. This is only at the start of the system.
        if (_recentFeePeriodsStorage(period - 1).startingDebtIndex == 0) return 0;

        uint closingDebtIndex = uint256(_recentFeePeriodsStorage(period - 1).startingDebtIndex).sub(1);

        uint ownershipPercentage;
        uint debtEntryIndex;
        (ownershipPercentage, debtEntryIndex) = feePoolState().applicableIssuanceData(account,
closingDebtIndex);

        // internal function will check closingDebtIndex has corresponding debtLedger entry
        return _effectiveDebtRatioForPeriod(closingDebtIndex, ownershipPercentage, debtEntryIndex);
    }

    /**
     * @notice Get the feePeriodID of the last claim this account made
     * @param _claimingAddress account to check the last fee period ID claim for
     * @return uint of the feePeriodID this account last claimed
     */
    function getLastFeeWithdrawal(address _claimingAddress) public view returns (uint) {
        return
feePoolEternalStorage().getUIntValue(keccak256(abi.encodePacked(LAST_FEE_WITHDRAWAL,
_claimingAddress)));
    }

    /**
     * @notice Calculate the collateral ratio before user is blocked from claiming.
     */
    function getPenaltyThresholdRatio() public view returns (uint) {
        return getIssuanceRatio().multiplyDecimal(SafeDecimalMath.unit().add(getTargetThreshold()));
    }

    /**
     * @notice Set the feePeriodID of the last claim this account made
     * @param _claimingAddress account to set the last feePeriodID claim for
     * @param _feePeriodID the feePeriodID this account claimed fees for
     */
    function _setLastFeeWithdrawal(address _claimingAddress, uint _feePeriodID) internal {
        feePoolEternalStorage().setUIntValue(
            keccak256(abi.encodePacked(LAST_FEE_WITHDRAWAL, _claimingAddress)),
            _feePeriodID
        );
```

```
    }

    /* ========== Modifiers ========== */
    modifier onlyInternalContracts {
        bool isExchanger = msg.sender == address(exchanger());
        bool isSynth = issuer().synthsByAddress(msg.sender) != bytes32(0);
        bool isEtherCollateralsUSD = msg.sender == address(etherCollateralsUSD());

        require(isExchanger || isSynth || isEtherCollateralsUSD, "Only Internal Contracts");
        _;
    }

    modifier onlyIssuer {
        require(msg.sender == address(issuer()), "FeePool: Only Issuer Authorised");
        _;
    }

    modifier notFeeAddress(address account) {
        require(account != FEE_ADDRESS, "Fee address not allowed");
        _;
    }

    modifier issuanceActive() {
        systemStatus().requireIssuanceActive();
        _;
    }

    /* ========== Proxy Events ========== */

    event IssuanceDebtRatioEntry(
        address indexed account,
        uint debtRatio,
        uint debtEntryIndex,
        uint feePeriodStartingDebtIndex
    );
    bytes32 private constant ISSUANCEDEBTRATIOENTRY_SIG = keccak256(
        "IssuanceDebtRatioEntry(address,uint256,uint256,uint256)"
    );

    function emitIssuanceDebtRatioEntry(
        address account,
        uint debtRatio,
        uint debtEntryIndex,
        uint feePeriodStartingDebtIndex
    ) internal {
        proxy._emit(
            abi.encode(debtRatio, debtEntryIndex, feePeriodStartingDebtIndex),
            2,
            ISSUANCEDEBTRATIOENTRY_SIG,
            bytes32(uint256(uint160(account))),
            0,
            0
        );
    }

    event FeePeriodClosed(uint feePeriodId);
    bytes32 private constant FEEPERIODCLOSED_SIG = keccak256("FeePeriodClosed(uint256)");

    function emitFeePeriodClosed(uint feePeriodId) internal {
        proxy._emit(abi.encode(feePeriodId), 1, FEEPERIODCLOSED_SIG, 0, 0, 0);
    }

    event FeesClaimed(address account, uint zUSDAmount, uint snxRewards);
    bytes32 private constant FEESCLAIMED_SIG = keccak256("FeesClaimed(address,uint256,uint256)");

    function emitFeesClaimed(
        address account,
        uint zUSDAmount,
        uint hznRewards
    ) internal {
        proxy._emit(abi.encode(account, zUSDAmount, hznRewards), 1, FEESCLAIMED_SIG, 0, 0, 0);
    }
}
```

FeePoolEternalStorage.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./EternalStorage.sol";
import "./LimitedSetup.sol";


// https://docs.synthetix.io/contracts/source/contracts/feepooleternalstorage
contract FeePoolEternalStorage is EternalStorage, LimitedSetup {
    bytes32 internal constant LAST_FEE_WITHDRAWAL = "last_fee_withdrawal";
```

```
    constructor(address _owner, address _feePool) public EternalStorage(_owner, _feePool) LimitedSetup(6 weeks) {}

    function importFeeWithdrawalData(address[] calldata accounts, uint[] calldata feePeriodIDs)
        external
        onlyOwner
        onlyDuringSetup
    {
        require(accounts.length == feePeriodIDs.length, "Length mismatch");

        for (uint8 i = 0; i < accounts.length; i++) {
            this.setUIntValue(keccak256(abi.encodePacked(LAST_FEE_WITHDRAWAL, accounts[i])), feePeriodIDs[i]);
        }
    }
}
```

FeePoolState.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./LimitedSetup.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IFeePool.sol";


// https://docs.synthetix.io/contracts/source/contracts/feepoolstate
contract FeePoolState is Owned, LimitedSetup {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    /* ========== STATE VARIABLES ========== */

    uint8 public constant FEE_PERIOD_LENGTH = 6;

    address public feePool;

    // The IssuanceData activity that's happened in a fee period.
    struct IssuanceData {
        uint debtPercentage;
        uint debtEntryIndex;
    }

    // The IssuanceData activity that's happened in a fee period.
    mapping(address => IssuanceData[FEE_PERIOD_LENGTH]) public accountIssuanceLedger;

    constructor(address _owner, IFeePool _feePool) public Owned(_owner) LimitedSetup(6 weeks) {
        feePool = address(_feePool);
    }

    /* ========== SETTERS ========== */

    /**
     * @notice set the FeePool contract as it is the only authority to be able to call
     * appendAccountIssuanceRecord with the onlyFeePool modifer
     * @dev Must be set by owner when FeePool logic is upgraded
     */
    function setFeePool(IFeePool _feePool) external onlyOwner {
        feePool = address(_feePool);
    }

    /* ========== VIEWS ========== */

    /**
     * @notice Get an accounts issuanceData for
     * @param account users account
     * @param index Index in the array to retrieve. Upto FEE_PERIOD_LENGTH
     */
    function getAccountsDebtEntry(address account, uint index)
        public
        view
        returns (uint debtPercentage, uint debtEntryIndex)
    {
        require(index < FEE_PERIOD_LENGTH, "index exceeds the FEE_PERIOD_LENGTH");

        debtPercentage = accountIssuanceLedger[account][index].debtPercentage;
        debtEntryIndex = accountIssuanceLedger[account][index].debtEntryIndex;
    }
```

```
    /**
     * @notice Find the oldest debtEntryIndex for the corresponding closingDebtIndex
     * @param account users account
     * @param closingDebtIndex the last periods debt index on close
     */
    function applicableIssuanceData(address account, uint closingDebtIndex) external view returns (uint, uint) {
        IssuanceData[FEE_PERIOD_LENGTH] memory issuanceData = accountIssuanceLedger[account];

        // We want to use the user's debtEntryIndex at when the period closed
        // Find the oldest debtEntryIndex for the corresponding closingDebtIndex
        for (uint i = 0; i < FEE_PERIOD_LENGTH; i++) {
            if (closingDebtIndex >= issuanceData[i].debtEntryIndex) {
                return (issuanceData[i].debtPercentage, issuanceData[i].debtEntryIndex);
            }
        }
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /**
     * @notice Logs an accounts issuance data in the current fee period which is then stored historically
     * @param account Message.Senders account address
     * @param debtRatio Debt of this account as a percentage of the global debt.
     * @param debtEntryIndex The index in the global debt ledger. synthetix.synthetixState().issuanceData(account)
     * @param currentPeriodStartDebtIndex The startingDebtIndex of the current fee period
     * @dev onlyFeePool to call me on synthetix.issue() & synthetix.burn() calls to store the locked HZN
     * per fee period so we know to allocate the correct proportions of fees and rewards per period
     accountIssuanceLedger[account][0] has the latest locked amount for the current period. This can be update as many time
     accountIssuanceLedger[account][1-2] has the last locked amount for a previous period they minted or burned
     */
    function appendAccountIssuanceRecord(
        address account,
        uint debtRatio,
        uint debtEntryIndex,
        uint currentPeriodStartDebtIndex
    ) external onlyFeePool {
        // Is the current debtEntryIndex within this fee period
        if (accountIssuanceLedger[account][0].debtEntryIndex < currentPeriodStartDebtIndex) {
            // If its older then shift the previous IssuanceData entries periods down to make room for the new one.
            issuanceDataIndexOrder(account);
        }

        // Always store the latest IssuanceData entry at [0]
        accountIssuanceLedger[account][0].debtPercentage = debtRatio;
        accountIssuanceLedger[account][0].debtEntryIndex = debtEntryIndex;
    }

    /**
     * @notice Pushes down the entire array of debt ratios per fee period
     */
    function issuanceDataIndexOrder(address account) private {
        for (uint i = FEE_PERIOD_LENGTH - 2; i < FEE_PERIOD_LENGTH; i--) {
            uint next = i + 1;
            accountIssuanceLedger[account][next].debtPercentage = accountIssuanceLedger[account][i].debtPercentage;
            accountIssuanceLedger[account][next].debtEntryIndex = accountIssuanceLedger[account][i].debtEntryIndex;
        }
    }

    /**
     * @notice Import issuer data from synthetixState.issuerData on FeePeriodClose() block #
     * @dev Only callable by the contract owner, and only for 6 weeks after deployment.
     * @param accounts Array of issuing addresses
     * @param ratios Array of debt ratios
     * @param periodToInsert The Fee Period to insert the historical records into
     * @param feePeriodCloseIndex An accounts debtEntryIndex is valid when within the fee peroid,
     * since the input ratio will be an average of the pervious periods it just needs to be
     * > recentFeePeriods[periodToInsert].startingDebtIndex
     * < recentFeePeriods[periodToInsert - 1].startingDebtIndex
     */
    function importIssuerData(
        address[] calldata accounts,
        uint[] calldata ratios,
        uint periodToInsert,
        uint feePeriodCloseIndex
    ) external onlyOwner onlyDuringSetup {
        require(accounts.length == ratios.length, "Length mismatch");

        for (uint i = 0; i < accounts.length; i++) {
            accountIssuanceLedger[accounts[i]][periodToInsert].debtPercentage = ratios[i];
            accountIssuanceLedger[accounts[i]][periodToInsert].debtEntryIndex = feePeriodCloseIndex;
            emit IssuanceDebtRatioEntry(accounts[i], ratios[i], feePeriodCloseIndex);
```

```
            }
        }

        /* ========== MODIFIERS ========== */

        modifier onlyFeePool {
            require(msg.sender == address(feePool), "Only the FeePool contract can perform this action");
            _;
        }

        /* ========== Events ========== */
        event IssuanceDebtRatioEntry(address indexed account, uint debtRatio, uint feePeriodCloseIndex);
}
```

FixedSupplySchedule.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./interfaces/ISupplySchedule.sol";

// Libraries
import "./SafeDecimalMath.sol";
import "./Math.sol";

// Internal references
import "./Proxy.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/fixedsupplyschedule
contract FixedSupplySchedule is Owned, MixinResolver, ISupplySchedule {
        using SafeMath for uint;
        using SafeDecimalMath for uint;
        using Math for uint;

        /* ========== CONSTANTS ========== */

        // Max HZN rewards for minter
        uint public constant MAX_MINTER_REWARD = 200 ether; //1 ether == 1e18

        // Default mintPeriodDuration
        uint public constant DEFAULT_MINT_PERIOD_DURATION = 1 weeks;
        // Default mintBuffer
        uint public constant DEFAULT_MINT_BUFFER = 1 days;

        /* ========== STORAGE VARIABLES ========== */

        // Point in time that the inflation starts from
        uint public inflationStartDate;
        // Time of the last inflation supply mint event
        uint public lastMintEvent;
        // Counter for number of minting periods since the start of supply inflation
        uint public mintPeriodCounter;
        // The duration of the period till the next minting occurs aka inflation/minting event frequency
        uint public mintPeriodDuration = DEFAULT_MINT_PERIOD_DURATION;
        // The buffer needs to be added so inflation is minted after feePeriod closes
        uint public mintBuffer = DEFAULT_MINT_BUFFER;
        // The periodic inflationary supply. Set in the constructor and fixed throughout the duration
        uint public fixedPeriodicSupply;
        // The period that the suply schedule ends
        uint public supplyEnd;
        // The number of HZN rewarded to the caller of Synthetix.mint()
        uint public minterReward;

        /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

        bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";

        bytes32[24] private addressesToCache = [CONTRACT_SYNTHETIX];

        constructor(
            address _owner,
            address _resolver,
            uint _inflationStartDate,
            uint _lastMintEvent,
            uint _mintPeriodCounter,
            uint _mintPeriodDuration,
            uint _mintBuffer,
            uint _fixedPeriodicSupply,
            uint _supplyEnd,
            uint _minterReward
        ) public Owned(_owner) MixinResolver(_resolver, addressesToCache) {
```

```
        // inflationStartDate: 0 defaults to current timestamp
        if (_inflationStartDate != 0) {
            inflationStartDate = _inflationStartDate;
        } else {
            inflationStartDate = block.timestamp;
        }
        // lastMintEvent: should be strictly greater than the infaltion start time (if not zero)
        // mintPeriodCounter: should not be zero iff lastMintEvent is not zero
        if (_lastMintEvent != 0) {
            require(_lastMintEvent > inflationStartDate, "Mint event can't happen before inflation starts");
            require(_mintPeriodCounter > 0, "At least a mint event has already occurred");
        }
        require(_mintBuffer <= _mintPeriodDuration, "Buffer can't be greater than period");
        require(_minterReward <= MAX_MINTER_REWARD, "Reward can't exceed max minter reward");

        lastMintEvent = _lastMintEvent;
        mintPeriodCounter = _mintPeriodCounter;
        fixedPeriodicSupply = _fixedPeriodicSupply;
        // mintBuffer: defaults to DEFAULT_MINT_BUFFER if zero
        if (_mintBuffer != 0) {
            mintBuffer = _mintBuffer;
        }
        // mintPeriodDuration: defaults to DEFAULT_MINT_PERIOD_DURATION if zero
        if (_mintPeriodDuration != 0) {
            mintPeriodDuration = _mintPeriodDuration;
        }
        supplyEnd = _supplyEnd;
        minterReward = _minterReward;
    }

    // ========== VIEWS ==========

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Synthetix address"));
    }

    /**
     * @return The amount of HZN mintable for the inflationary supply
     */
    function mintableSupply() external view returns (uint) {
        uint totalAmount;

        if (!isMintable() || fixedPeriodicSupply == 0) {
            return 0;
        }

        uint remainingPeriodsToMint = periodsSinceLastIssuance();

        uint currentPeriod = mintPeriodCounter;

        // Calculate total mintable supply
        // The function stops after supplyEnd
        while (remainingPeriodsToMint > 0) {
            currentPeriod = currentPeriod.add(1);

            if (currentPeriod < supplyEnd) {
                // If current period is before supply end we add the fixed supply to mintableSupply
                totalAmount = totalAmount.add(fixedPeriodicSupply);
            } else {
                // break the loop if the infation has reached its end
                break;
            }

            remainingPeriodsToMint--;
        }

        return totalAmount;
    }

    /**
     * @dev Take timeDiff in seconds (Dividend) and mintPeriodDuration as (Divisor)
     * @return Calculate the number of minting periods since last mint rounded down
     */
    function periodsSinceLastIssuance() public view returns (uint) {
        // Get minting periods since lastMintEvent
        // If lastMintEvent not set or 0, then start from inflation start date.
        uint timeDiff = lastMintEvent > 0 ? block.timestamp.sub(lastMintEvent) :
block.timestamp.sub(inflationStartDate);
        return timeDiff.div(mintPeriodDuration);
    }

    /**
     * @return boolean whether the mintPeriodDuration (default is 7 days)
     * has passed since the lastMintEvent.
     * */
    function isMintable() public view returns (bool) {
        if (block.timestamp - lastMintEvent > mintPeriodDuration) {
```

```
            return true;
        }
        return false;
    }

    // ========== MUTATIVE FUNCTIONS ==========

    /**
     * @notice Record the mint event from Synthetix by incrementing the inflation
     * period counter for the number of periods minted (probabaly always 1)
     * and store the time of the event.
     * @param supplyMinted the amount of HZN the total supply was inflated by.
     * */
    function recordMintEvent(uint supplyMinted) external onlySynthetix returns (bool) {
        uint numberOfPeriodsIssued = periodsSinceLastIssuance();

        // add number of periods minted to mintPeriodCounter
        mintPeriodCounter = mintPeriodCounter.add(numberOfPeriodsIssued);

        // Update mint event to latest period issued (start date + number of periods issued * seconds in a period)
        // A time buffer is added so inflation is minted after feePeriod closes
        lastMintEvent = inflationStartDate.add(mintPeriodCounter.mul(mintPeriodDuration)).add(mintBuffer);

        emit SupplyMinted(supplyMinted, numberOfPeriodsIssued, lastMintEvent, block.timestamp);
        return true;
    }

    // ========== SETTERS ========== */

    /**
     * @notice Sets the reward amount of HZN for the caller of the public
     * function Synthetix.mint().
     * This incentivises anyone to mint the inflationary supply and the mintr
     * Reward will be deducted from the inflationary supply and sent to the caller.
     * @param amount the amount of HZN to reward the minter.
     * */
    function setMinterReward(uint amount) external onlyOwner {
        require(amount <= MAX_MINTER_REWARD, "Reward can't exceed max minter reward");
        minterReward = amount;
        emit MinterRewardUpdated(minterReward);
    }

    // ========== MODIFIERS ==========

    /**
     * @notice Only the Synthetix contract is authorised to call this function
     * */
    modifier onlySynthetix() {
        require(msg.sender == address(synthetix()), "SupplySchedule: Only the synthetix contract can perform this action");
        _;
    }

    /* ========== EVENTS ========== */
    /**
     * @notice Emitted when the inflationary supply is minted
     * */
    event SupplyMinted(uint supplyMinted, uint numberOfPeriodsIssued, uint lastMintEvent, uint timestamp);

    /**
     * @notice Emitted when the HZN minter reward amount is updated
     * */
    event MinterRewardUpdated(uint newRewardAmount);
}
```

FlexibleStorage.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./ContractStorage.sol";
import "./interfaces/IFlexibleStorage.sol";

// Internal References
import "./interfaces/IAddressResolver.sol";


// https://docs.synthetix.io/contracts/source/contracts/flexiblestorage
contract FlexibleStorage is ContractStorage, IFlexibleStorage {
    mapping(bytes32 => mapping(bytes32 => uint)) internal uintStorage;
    mapping(bytes32 => mapping(bytes32 => int)) internal intStorage;
    mapping(bytes32 => mapping(bytes32 => address)) internal addressStorage;
    mapping(bytes32 => mapping(bytes32 => bool)) internal boolStorage;
    mapping(bytes32 => mapping(bytes32 => bytes32)) internal bytes32Storage;

    constructor(address _resolver) public ContractStorage(_resolver) {}
```

```
    /* ========== INTERNAL FUNCTIONS ========== */

    function _setUIntValue(
        bytes32 contractName,
        bytes32 record,
        uint value
    ) internal {
        uintStorage[_memoizeHash(contractName)][record] = value;
        emit ValueSetUInt(contractName, record, value);
    }

    function _setIntValue(
        bytes32 contractName,
        bytes32 record,
        int value
    ) internal {
        intStorage[_memoizeHash(contractName)][record] = value;
        emit ValueSetInt(contractName, record, value);
    }

    function _setAddressValue(
        bytes32 contractName,
        bytes32 record,
        address value
    ) internal {
        addressStorage[_memoizeHash(contractName)][record] = value;
        emit ValueSetAddress(contractName, record, value);
    }

    function _setBoolValue(
        bytes32 contractName,
        bytes32 record,
        bool value
    ) internal {
        boolStorage[_memoizeHash(contractName)][record] = value;
        emit ValueSetBool(contractName, record, value);
    }

    function _setBytes32Value(
        bytes32 contractName,
        bytes32 record,
        bytes32 value
    ) internal {
        bytes32Storage[_memoizeHash(contractName)][record] = value;
        emit ValueSetBytes32(contractName, record, value);
    }

    /* ========== VIEWS ========== */

    function getUIntValue(bytes32 contractName, bytes32 record) external view returns (uint) {
        return uintStorage[hashes[contractName]][record];
    }

    function getUIntValues(bytes32 contractName, bytes32[] calldata records) external view returns (uint[] memory) {
        uint[] memory results = new uint[](records.length);

        mapping(bytes32 => uint) storage data = uintStorage[hashes[contractName]];
        for (uint i = 0; i < records.length; i++) {
            results[i] = data[records[i]];
        }
        return results;
    }

    function getIntValue(bytes32 contractName, bytes32 record) external view returns (int) {
        return intStorage[hashes[contractName]][record];
    }

    function getIntValues(bytes32 contractName, bytes32[] calldata records) external view returns (int[] memory) {
        int[] memory results = new int[](records.length);

        mapping(bytes32 => int) storage data = intStorage[hashes[contractName]];
        for (uint i = 0; i < records.length; i++) {
            results[i] = data[records[i]];
        }
        return results;
    }

    function getAddressValue(bytes32 contractName, bytes32 record) external view returns (address) {
        return addressStorage[hashes[contractName]][record];
    }

    function getAddressValues(bytes32 contractName, bytes32[] calldata records) external view returns (address[] memory) {
        address[] memory results = new address[](records.length);
```

```
        mapping(bytes32 => address) storage data = addressStorage[hashes[contractName]];
        for (uint i = 0; i < records.length; i++) {
            results[i] = data[records[i]];
        }
        return results;
    }

    function getBoolValue(bytes32 contractName, bytes32 record) external view returns (bool) {
        return boolStorage[hashes[contractName]][record];
    }

    function getBoolValues(bytes32 contractName, bytes32[] calldata records) external view returns (bool[] memory) {
        bool[] memory results = new bool[](records.length);

        mapping(bytes32 => bool) storage data = boolStorage[hashes[contractName]];
        for (uint i = 0; i < records.length; i++) {
            results[i] = data[records[i]];
        }
        return results;
    }

    function getBytes32Value(bytes32 contractName, bytes32 record) external view returns (bytes32) {
        return bytes32Storage[hashes[contractName]][record];
    }

    function getBytes32Values(bytes32 contractName, bytes32[] calldata records) external view returns (bytes32[] memory) {
        bytes32[] memory results = new bytes32[](records.length);

        mapping(bytes32 => bytes32) storage data = bytes32Storage[hashes[contractName]];
        for (uint i = 0; i < records.length; i++) {
            results[i] = data[records[i]];
        }
        return results;
    }

    /* ========== RESTRICTED FUNCTIONS ========== */
    function setUIntValue(
        bytes32 contractName,
        bytes32 record,
        uint value
    ) external onlyContract(contractName) {
        _setUIntValue(contractName, record, value);
    }

    function setUIntValues(
        bytes32 contractName,
        bytes32[] calldata records,
        uint[] calldata values
    ) external onlyContract(contractName) {
        require(records.length == values.length, "Input lengths must match");

        for (uint i = 0; i < records.length; i++) {
            _setUIntValue(contractName, records[i], values[i]);
        }
    }

    function deleteUIntValue(bytes32 contractName, bytes32 record) external onlyContract(contractName) {
        uint value = uintStorage[hashes[contractName]][record];
        emit ValueDeletedUInt(contractName, record, value);
        delete uintStorage[hashes[contractName]][record];
    }

    function setIntValue(
        bytes32 contractName,
        bytes32 record,
        int value
    ) external onlyContract(contractName) {
        _setIntValue(contractName, record, value);
    }

    function setIntValues(
        bytes32 contractName,
        bytes32[] calldata records,
        int[] calldata values
    ) external onlyContract(contractName) {
        require(records.length == values.length, "Input lengths must match");

        for (uint i = 0; i < records.length; i++) {
            _setIntValue(contractName, records[i], values[i]);
        }
    }

    function deleteIntValue(bytes32 contractName, bytes32 record) external onlyContract(contractName) {
        int value = intStorage[hashes[contractName]][record];
```

```
        emit ValueDeletedInt(contractName, record, value);
        delete intStorage[hashes[contractName]][record];
    }

    function setAddressValue(
        bytes32 contractName,
        bytes32 record,
        address value
    ) external onlyContract(contractName) {
        _setAddressValue(contractName, record, value);
    }

    function setAddressValues(
        bytes32 contractName,
        bytes32[] calldata records,
        address[] calldata values
    ) external onlyContract(contractName) {
        require(records.length == values.length, "Input lengths must match");

        for (uint i = 0; i < records.length; i++) {
            _setAddressValue(contractName, records[i], values[i]);
        }
    }

    function deleteAddressValue(bytes32 contractName, bytes32 record) external onlyContract(contractName) {
        address value = addressStorage[hashes[contractName]][record];
        emit ValueDeletedAddress(contractName, record, value);
        delete addressStorage[hashes[contractName]][record];
    }

    function setBoolValue(
        bytes32 contractName,
        bytes32 record,
        bool value
    ) external onlyContract(contractName) {
        _setBoolValue(contractName, record, value);
    }

    function setBoolValues(
        bytes32 contractName,
        bytes32[] calldata records,
        bool[] calldata values
    ) external onlyContract(contractName) {
        require(records.length == values.length, "Input lengths must match");

        for (uint i = 0; i < records.length; i++) {
            _setBoolValue(contractName, records[i], values[i]);
        }
    }

    function deleteBoolValue(bytes32 contractName, bytes32 record) external onlyContract(contractName) {
        bool value = boolStorage[hashes[contractName]][record];
        emit ValueDeletedBool(contractName, record, value);
        delete boolStorage[hashes[contractName]][record];
    }

    function setBytes32Value(
        bytes32 contractName,
        bytes32 record,
        bytes32 value
    ) external onlyContract(contractName) {
        _setBytes32Value(contractName, record, value);
    }

    function setBytes32Values(
        bytes32 contractName,
        bytes32[] calldata records,
        bytes32[] calldata values
    ) external onlyContract(contractName) {
        require(records.length == values.length, "Input lengths must match");

        for (uint i = 0; i < records.length; i++) {
            _setBytes32Value(contractName, records[i], values[i]);
        }
    }

    function deleteBytes32Value(bytes32 contractName, bytes32 record) external onlyContract(contractName) {
        bytes32 value = bytes32Storage[hashes[contractName]][record];
        emit ValueDeletedBytes32(contractName, record, value);
        delete bytes32Storage[hashes[contractName]][record];
    }

    /* ========== EVENTS ========== */

    event ValueSetUInt(bytes32 contractName, bytes32 record, uint value);
    event ValueDeletedUInt(bytes32 contractName, bytes32 record, uint value);
```

```
    event ValueSetInt(bytes32 contractName, bytes32 record, int value);
    event ValueDeletedInt(bytes32 contractName, bytes32 record, int value);

    event ValueSetAddress(bytes32 contractName, bytes32 record, address value);
    event ValueDeletedAddress(bytes32 contractName, bytes32 record, address value);

    event ValueSetBool(bytes32 contractName, bytes32 record, bool value);
    event ValueDeletedBool(bytes32 contractName, bytes32 record, bool value);

    event ValueSetBytes32(bytes32 contractName, bytes32 record, bytes32 value);
    event ValueDeletedBytes32(bytes32 contractName, bytes32 record, bytes32 value);
}
```

IAddressResolver.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iaddressresolver
interface IAddressResolver {
    function getAddress(bytes32 name) external view returns (address);

    function getSynth(bytes32 key) external view returns (address);

    function requireAndGetAddress(bytes32 name, string calldata reason) external view returns (address);
}
```

IBinaryOption.sol

```
pragma solidity >=0.4.24;

import "../interfaces/IBinaryOptionMarket.sol";
import "../interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/interfaces/ibinaryoption
interface IBinaryOption {
    /* ========== VIEWS / VARIABLES ========== */

    function market() external view returns (IBinaryOptionMarket);

    function bidOf(address account) external view returns (uint);

    function totalBids() external view returns (uint);

    function balanceOf(address account) external view returns (uint);

    function totalSupply() external view returns (uint);

    function claimableBalanceOf(address account) external view returns (uint);

    function totalClaimableSupply() external view returns (uint);
}
```

IBinaryOptionMarket.sol

```
pragma solidity >=0.4.24;

import "../interfaces/IBinaryOptionMarketManager.sol";
import "../interfaces/IBinaryOption.sol";


// https://docs.synthetix.io/contracts/source/interfaces/ibinaryoptionmarket
interface IBinaryOptionMarket {
    /* ========== TYPES ========== */

    enum Phase {Bidding, Trading, Maturity, Expiry}
    enum Side {Long, Short}

    /* ========== VIEWS / VARIABLES ========== */

    function options() external view returns (IBinaryOption long, IBinaryOption short);

    function prices() external view returns (uint long, uint short);

    function times()
        external
        view
        returns (
            uint biddingEnd,
            uint maturity,
            uint destructino
        );
```

```
    function oracleDetails()
        external
        view
        returns (
            bytes32 key,
            uint strikePrice,
            uint finalPrice
        );

    function fees()
        external
        view
        returns (
            uint poolFee,
            uint creatorFee,
            uint refundFee
        );

    function creatorLimits() external view returns (uint capitalRequirement, uint skewLimit);

    function deposited() external view returns (uint);

    function creator() external view returns (address);

    function resolved() external view returns (bool);

    function refundsEnabled() external view returns (bool);

    function phase() external view returns (Phase);

    function oraclePriceAndTimestamp() external view returns (uint price, uint updatedAt);

    function canResolve() external view returns (bool);

    function result() external view returns (Side);

    function pricesAfterBidOrRefund(
        Side side,
        uint value,
        bool refund
    ) external view returns (uint long, uint short);

    function bidOrRefundForPrice(
        Side bidSide,
        Side priceSide,
        uint price,
        bool refund
    ) external view returns (uint);

    function bidsOf(address account) external view returns (uint long, uint short);

    function totalBids() external view returns (uint long, uint short);

    function claimableBalancesOf(address account) external view returns (uint long, uint short);

    function totalClaimableSupplies() external view returns (uint long, uint short);

    function balancesOf(address account) external view returns (uint long, uint short);

    function totalSupplies() external view returns (uint long, uint short);

    function exercisableDeposits() external view returns (uint);

    /* ========== MUTATIVE FUNCTIONS ========== */

    function bid(Side side, uint value) external;

    function refund(Side side, uint value) external returns (uint refundMinusFee);

    function claimOptions() external returns (uint longClaimed, uint shortClaimed);

    function exerciseOptions() external returns (uint);
}
```

IBinaryOptionMarketManager.sol

```
pragma solidity >=0.4.24;

import "../interfaces/IBinaryOptionMarket.sol";


// https://docs.synthetix.io/contracts/source/interfaces/ibinaryoptionmarketmanager
interface IBinaryOptionMarketManager {
    /* ========== VIEWS / VARIABLES ========== */

    function fees()
```

```
        external
        view
        returns (
            uint poolFee,
            uint creatorFee,
            uint refundFee
        );

    function durations()
        external
        view
        returns (
            uint maxOraclePriceAge,
            uint expiryDuration,
            uint maxTimeToMaturity
        );

    function creatorLimits() external view returns (uint capitalRequirement, uint skewLimit);

    function marketCreationEnabled() external view returns (bool);

    function totalDeposited() external view returns (uint);

    function numActiveMarkets() external view returns (uint);

    function activeMarkets(uint index, uint pageSize) external view returns (address[] memory);

    function numMaturedMarkets() external view returns (uint);

    function maturedMarkets(uint index, uint pageSize) external view returns (address[] memory);

    /* ========== MUTATIVE FUNCTIONS ========== */

    function createMarket(
        bytes32 oracleKey,
        uint strikePrice,
        bool refundsEnabled,
        uint[2] calldata times, // [biddingEnd, maturity]
        uint[2] calldata bids // [longBid, shortBid]
    ) external returns (IBinaryOptionMarket);

    function resolveMarket(address market) external;

    function cancelMarket(address market) external;

    function expireMarkets(address[] calldata market) external;
}
```

IDebtCache.sol

```
pragma solidity >=0.4.24;

import "../interfaces/ISynth.sol";


// https://docs.synthetix.io/contracts/source/interfaces/idebtcache
interface IDebtCache {
    // Views

    function cachedDebt() external view returns (uint);

    function cachedSynthDebt(bytes32 currencyKey) external view returns (uint);

    function cacheTimestamp() external view returns (uint);

    function cacheInvalid() external view returns (bool);

    function cacheStale() external view returns (bool);

    function currentSynthDebts(bytes32[] calldata currencyKeys)
        external
        view
        returns (uint[] memory debtValues, bool anyRateIsInvalid);

    function cachedSynthDebts(bytes32[] calldata currencyKeys) external view returns (uint[] memory
debtValues);

    function currentDebt() external view returns (uint debt, bool anyRateIsInvalid);

    function cacheInfo()
        external
        view
        returns (
            uint debt,
            uint timestamp,
            bool isInvalid,
```

```
            bool isStale
        );

    // Mutative functions

    function takeDebtSnapshot() external;

    function updateCachedSynthDebts(bytes32[] calldata currencyKeys) external;
}
```

IDelegateApprovals.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/idelegateapprovals
interface IDelegateApprovals {
    // Views
    function canBurnFor(address authoriser, address delegate) external view returns (bool);

    function canIssueFor(address authoriser, address delegate) external view returns (bool);

    function canClaimFor(address authoriser, address delegate) external view returns (bool);

    function canExchangeFor(address authoriser, address delegate) external view returns (bool);

    // Mutative
    function approveAllDelegatePowers(address delegate) external;

    function removeAllDelegatePowers(address delegate) external;

    function approveBurnOnBehalf(address delegate) external;

    function removeBurnOnBehalf(address delegate) external;

    function approveIssueOnBehalf(address delegate) external;

    function removeIssueOnBehalf(address delegate) external;

    function approveClaimOnBehalf(address delegate) external;

    function removeClaimOnBehalf(address delegate) external;

    function approveExchangeOnBehalf(address delegate) external;

    function removeExchangeOnBehalf(address delegate) external;
}
```

IDepot.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/idepot
interface IDepot {
    // Views
    function fundsWallet() external view returns (address payable);

    function maxEthPurchase() external view returns (uint);

    function minimumDepositAmount() external view returns (uint);

    function synthsReceivedForEther(uint amount) external view returns (uint);

    function totalSellableDeposits() external view returns (uint);

    // Mutative functions
    function depositSynths(uint amount) external;

    function exchangeEtherForSynths() external payable returns (uint);

    function exchangeEtherForSynthsAtRate(uint guaranteedRate) external payable returns (uint);

    function withdrawMyDepositedSynths() external;

    // Note: On mainnet no SNX has been deposited. The following functions are kept alive for testnet SNX faucets.
    function exchangeEtherForSNX() external payable returns (uint);

    function exchangeEtherForSNXAtRate(uint guaranteedRate, uint guaranteedSynthetixRate) external payable
returns (uint);

    function exchangeSynthsForSNX(uint synthAmount) external returns (uint);

    function synthetixReceivedForEther(uint amount) external view returns (uint);
```

```
        function synthetixReceivedForSynths(uint amount) external view returns (uint);

        function withdrawSynthetix(uint amount) external;
}
```

IERC20.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/ierc20
interface IERC20 {
        // ERC20 Optional Views
        function name() external view returns (string memory);

        function symbol() external view returns (string memory);

        function decimals() external view returns (uint8);

        // Views
        function totalSupply() external view returns (uint);

        function balanceOf(address owner) external view returns (uint);

        function allowance(address owner, address spender) external view returns (uint);

        // Mutative functions
        function transfer(address to, uint value) external returns (bool);

        function approve(address spender, uint value) external returns (bool);

        function transferFrom(
                address from,
                address to,
                uint value
        ) external returns (bool);

        // Events
        event Transfer(address indexed from, address indexed to, uint value);

        event Approval(address indexed owner, address indexed spender, uint value);
}
```

IEtherCollateral.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iethercollateral
interface IEtherCollateral {
        // Views
        function totalIssuedSynths() external view returns (uint256);

        function totalLoansCreated() external view returns (uint256);

        function totalOpenLoanCount() external view returns (uint256);

        // Mutative functions
        function openLoan() external payable returns (uint256 loanID);

        function closeLoan(uint256 loanID) external;

        function liquidateUnclosedLoan(address _loanCreatorsAddress, uint256 _loanID) external;
}
```

IEtherCollateralsUSD.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iethercollateralsusd
interface IEtherCollateralsUSD {
        // Views
        function totalIssuedSynths() external view returns (uint256);

        function totalLoansCreated() external view returns (uint256);

        function totalOpenLoanCount() external view returns (uint256);

        // Mutative functions
        function openLoan(uint256 _loanAmount) external payable returns (uint256 loanID);

        function closeLoan(uint256 loanID) external;
```

```
    function liquidateUnclosedLoan(address _loanCreatorsAddress, uint256 _loanID) external;

    function depositCollateral(address account, uint256 loanID) external payable;

    function withdrawCollateral(uint256 loanID, uint256 withdrawAmount) external;

    function repayLoan(
        address _loanCreatorsAddress,
        uint256 _loanID,
        uint256 _repayAmount
    ) external;
}
```

IExchanger.sol

```
pragma solidity >=0.4.24;

import "./IVirtualSynth.sol";


// https://docs.synthetix.io/contracts/source/interfaces/iexchanger
interface IExchanger {
    // Views
    function calculateAmountAfterSettlement(
        address from,
        bytes32 currencyKey,
        uint amount,
        uint refunded
    ) external view returns (uint amountAfterSettlement);

    function isSynthRateInvalid(bytes32 currencyKey) external view returns (bool);

    function maxSecsLeftInWaitingPeriod(address account, bytes32 currencyKey) external view returns (uint);

    function settlementOwing(address account, bytes32 currencyKey)
        external
        view
        returns (
            uint reclaimAmount,
            uint rebateAmount,
            uint numEntries
        );

    function hasWaitingPeriodOrSettlementOwing(address account, bytes32 currencyKey) external view returns
(bool);

    function feeRateForExchange(bytes32 sourceCurrencyKey, bytes32 destinationCurrencyKey)
        external
        view
        returns (uint exchangeFeeRate);

    function getAmountsForExchange(
        uint sourceAmount,
        bytes32 sourceCurrencyKey,
        bytes32 destinationCurrencyKey
    )
        external
        view
        returns (
            uint amountReceived,
            uint fee,
            uint exchangeFeeRate
        );

    function priceDeviationThresholdFactor() external view returns (uint);

    function waitingPeriodSecs() external view returns (uint);

    // Mutative functions
    function exchange(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress
    ) external returns (uint amountReceived);

    function exchangeOnBehalf(
        address exchangeForAddress,
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external returns (uint amountReceived);

    function exchangeWithTracking(
```

```
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress,
        address originator,
        bytes32 trackingCode
    ) external returns (uint amountReceived);

    function exchangeOnBehalfWithTracking(
        address exchangeForAddress,
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external returns (uint amountReceived);

    function exchangeWithVirtual(
        address from,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address destinationAddress,
        bytes32 trackingCode
    ) external returns (uint amountReceived, IVirtualSynth vSynth);

    function settle(address from, bytes32 currencyKey)
        external
        returns (
            uint reclaimed,
            uint refunded,
            uint numEntries
        );

    function setLastExchangeRateForSynth(bytes32 currencyKey, uint rate) external;

    function suspendSynthWithInvalidRate(bytes32 currencyKey) external;
}
```

IExchangeRates.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iexchangerates
interface IExchangeRates {
    // Structs
    struct RateAndUpdatedTime {
        uint216 rate;
        uint40 time;
    }

    struct InversePricing {
        uint entryPoint;
        uint upperLimit;
        uint lowerLimit;
        bool frozenAtUpperLimit;
        bool frozenAtLowerLimit;
    }

    // Views
    function aggregators(bytes32 currencyKey) external view returns (address);

    function aggregatorWarningFlags() external view returns (address);

    function anyRateIsInvalid(bytes32[] calldata currencyKeys) external view returns (bool);

    function canFreezeRate(bytes32 currencyKey) external view returns (bool);

    function currentRoundForRate(bytes32 currencyKey) external view returns (uint);

    function currenciesUsingAggregator(address aggregator) external view returns (bytes32[] memory);

    function effectiveValue(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external view returns (uint value);

    function effectiveValueAndRates(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    )
```

```
        external
        view
        returns (
            uint value,
            uint sourceRate,
            uint destinationRate
        );

    function effectiveValueAtRound(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        uint roundIdForSrc,
        uint roundIdForDest
    ) external view returns (uint value);

    function getCurrentRoundId(bytes32 currencyKey) external view returns (uint);

    function getLastRoundIdBeforeElapsedSecs(
        bytes32 currencyKey,
        uint startingRoundId,
        uint startingTimestamp,
        uint timediff
    ) external view returns (uint);

    function inversePricing(bytes32 currencyKey)
        external
        view
        returns (
            uint entryPoint,
            uint upperLimit,
            uint lowerLimit,
            bool frozenAtUpperLimit,
            bool frozenAtLowerLimit
        );

    function lastRateUpdateTimes(bytes32 currencyKey) external view returns (uint256);

    function oracle() external view returns (address);

    function rateAndTimestampAtRound(bytes32 currencyKey, uint roundId) external view returns (uint rate, uint
time);

    function rateAndUpdatedTime(bytes32 currencyKey) external view returns (uint rate, uint time);

    function rateAndInvalid(bytes32 currencyKey) external view returns (uint rate, bool isInvalid);

    function rateForCurrency(bytes32 currencyKey) external view returns (uint);

    function rateIsFlagged(bytes32 currencyKey) external view returns (bool);

    function rateIsFrozen(bytes32 currencyKey) external view returns (bool);

    function rateIsInvalid(bytes32 currencyKey) external view returns (bool);

    function rateIsStale(bytes32 currencyKey) external view returns (bool);

    function rateStalePeriod() external view returns (uint);

    function ratesAndUpdatedTimeForCurrencyLastNRounds(bytes32 currencyKey, uint numRounds)
        external
        view
        returns (uint[] memory rates, uint[] memory times);

    function ratesAndInvalidForCurrencies(bytes32[] calldata currencyKeys)
        external
        view
        returns (uint[] memory rates, bool anyRateInvalid);

    function ratesForCurrencies(bytes32[] calldata currencyKeys) external view returns (uint[] memory);

    // Mutative functions
    function freezeRate(bytes32 currencyKey) external;
}
```

IExchangeState.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iexchangestate
interface IExchangeState {
    // Views
    struct ExchangeEntry {
        bytes32 src;
        uint amount;
```

```
        bytes32 dest;
        uint amountReceived;
        uint exchangeFeeRate;
        uint timestamp;
        uint roundIdForSrc;
        uint roundIdForDest;
    }

    function getLengthOfEntries(address account, bytes32 currencyKey) external view returns (uint);

    function getEntryAt(
        address account,
        bytes32 currencyKey,
        uint index
    )
        external
        view
        returns (
            bytes32 src,
            uint amount,
            bytes32 dest,
            uint amountReceived,
            uint exchangeFeeRate,
            uint timestamp,
            uint roundIdForSrc,
            uint roundIdForDest
        );

    function getMaxTimestamp(address account, bytes32 currencyKey) external view returns (uint);

    // Mutative functions
    function appendExchangeEntry(
        address account,
        bytes32 src,
        uint amount,
        bytes32 dest,
        uint amountReceived,
        uint exchangeFeeRate,
        uint timestamp,
        uint roundIdForSrc,
        uint roundIdForDest
    ) external;

    function removeEntries(address account, bytes32 currencyKey) external;
}
```

IFeePool.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/ifeepool
interface IFeePool {
    // Views

    // solhint-disable-next-line func-name-mixedcase
    function FEE_ADDRESS() external view returns (address);

    function feesAvailable(address account) external view returns (uint, uint);

    function feePeriodDuration() external view returns (uint);

    function isFeesClaimable(address account) external view returns (bool);

    function targetThreshold() external view returns (uint);

    function totalFeesAvailable() external view returns (uint);

    function totalRewardsAvailable() external view returns (uint);

    // Mutative Functions
    function claimFees() external returns (bool);

    function claimOnBehalf(address claimingForAddress) external returns (bool);

    function closeCurrentFeePeriod() external;

    // Restricted: used internally to Synthetix
    function appendAccountIssuanceRecord(
        address account,
        uint lockedAmount,
        uint debtEntryIndex
    ) external;

    function recordFeePaid(uint sUSDAmount) external;
```

```
    function setRewardsToDistribute(uint amount) external;
}
```

IFlexibleStorage.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iflexiblestorage
interface IFlexibleStorage {
    // Views
    function getUIntValue(bytes32 contractName, bytes32 record) external view returns (uint);

    function getUIntValues(bytes32 contractName, bytes32[] calldata records) external view returns (uint[] memory);

    function getIntValue(bytes32 contractName, bytes32 record) external view returns (int);

    function getIntValues(bytes32 contractName, bytes32[] calldata records) external view returns (int[] memory);

    function getAddressValue(bytes32 contractName, bytes32 record) external view returns (address);

    function getAddressValues(bytes32 contractName, bytes32[] calldata records) external view returns (address[] memory);

    function getBoolValue(bytes32 contractName, bytes32 record) external view returns (bool);

    function getBoolValues(bytes32 contractName, bytes32[] calldata records) external view returns (bool[] memory);

    function getBytes32Value(bytes32 contractName, bytes32 record) external view returns (bytes32);

    function getBytes32Values(bytes32 contractName, bytes32[] calldata records) external view returns (bytes32[] memory);

    // Mutative functions
    function deleteUIntValue(bytes32 contractName, bytes32 record) external;

    function deleteIntValue(bytes32 contractName, bytes32 record) external;

    function deleteAddressValue(bytes32 contractName, bytes32 record) external;

    function deleteBoolValue(bytes32 contractName, bytes32 record) external;

    function deleteBytes32Value(bytes32 contractName, bytes32 record) external;

    function setUIntValue(
        bytes32 contractName,
        bytes32 record,
        uint value
    ) external;

    function setUIntValues(
        bytes32 contractName,
        bytes32[] calldata records,
        uint[] calldata values
    ) external;

    function setIntValue(
        bytes32 contractName,
        bytes32 record,
        int value
    ) external;

    function setIntValues(
        bytes32 contractName,
        bytes32[] calldata records,
        int[] calldata values
    ) external;

    function setAddressValue(
        bytes32 contractName,
        bytes32 record,
        address value
    ) external;

    function setAddressValues(
        bytes32 contractName,
        bytes32[] calldata records,
        address[] calldata values
    ) external;

    function setBoolValue(
        bytes32 contractName,
        bytes32 record,
        bool value
```

```
    ) external;

    function setBoolValues(
        bytes32 contractName,
        bytes32[] calldata records,
        bool[] calldata values
    ) external;

    function setBytes32Value(
        bytes32 contractName,
        bytes32 record,
        bytes32 value
    ) external;

    function setBytes32Values(
        bytes32 contractName,
        bytes32[] calldata records,
        bytes32[] calldata values
    ) external;
}
```

IHasBalance.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/ihasbalance
interface IHasBalance {
    // Views
    function balanceOf(address account) external view returns (uint);
}
```

IIssuer.sol

```
pragma solidity >=0.4.24;

import "../interfaces/ISynth.sol";


// https://docs.synthetix.io/contracts/source/interfaces/iissuer
interface IIssuer {
    // Views
    function anySynthOrSNXRateIsInvalid() external view returns (bool anyRateInvalid);

    function availableCurrencyKeys() external view returns (bytes32[] memory);

    function availableSynthCount() external view returns (uint);

    function availableSynths(uint index) external view returns (ISynth);

    function canBurnSynths(address account) external view returns (bool);

    function collateral(address account) external view returns (uint);

    function collateralisationRatio(address issuer) external view returns (uint);

    function collateralisationRatioAndAnyRatesInvalid(address _issuer)
        external
        view
        returns (uint cratio, bool anyRateIsInvalid);

    function debtBalanceOf(address issuer, bytes32 currencyKey) external view returns (uint debtBalance);

    function issuanceRatio() external view returns (uint);

    function lastIssueEvent(address account) external view returns (uint);

    function maxIssuableSynths(address issuer) external view returns (uint maxIssuable);

    function minimumStakeTime() external view returns (uint);

    function remainingIssuableSynths(address issuer)
        external
        view
        returns (
            uint maxIssuable,
            uint alreadyIssued,
            uint totalSystemDebt
        );

    function synths(bytes32 currencyKey) external view returns (ISynth);

    function getSynths(bytes32[] calldata currencyKeys) external view returns (ISynth[] memory);

    function synthsByAddress(address synthAddress) external view returns (bytes32);
```

```
    function totalIssuedSynths(bytes32 currencyKey, bool excludeEtherCollateral) external view returns (uint);

    function transferableSynthetixAndAnyRateIsInvalid(address account, uint balance)
        external
        view
        returns (uint transferable, bool anyRateIsInvalid);

    // Restricted: used internally to Synthetix
    function issueSynths(address from, uint amount) external;

    function issueSynthsOnBehalf(
        address issueFor,
        address from,
        uint amount
    ) external;

    function issueMaxSynths(address from) external;

    function issueMaxSynthsOnBehalf(address issueFor, address from) external;

    function burnSynths(address from, uint amount) external;

    function burnSynthsOnBehalf(
        address burnForAddress,
        address from,
        uint amount
    ) external;

    function burnSynthsToTarget(address from) external;

    function burnSynthsToTargetOnBehalf(address burnForAddress, address from) external;

    function liquidateDelinquentAccount(
        address account,
        uint susdAmount,
        address liquidator
    ) external returns (uint totalRedeemed, uint amountToLiquidate);
}
```

ILiquidations.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/iliquidations
interface ILiquidations {
    // Views
    function isOpenForLiquidation(address account) external view returns (bool);

    function getLiquidationDeadlineForAccount(address account) external view returns (uint);

    function isLiquidationDeadlinePassed(address account) external view returns (bool);

    function liquidationDelay() external view returns (uint);

    function liquidationRatio() external view returns (uint);

    function liquidationPenalty() external view returns (uint);

    function calculateAmountToFixCollateral(uint debtBalance, uint collateral) external view returns (uint);

    // Mutative Functions
    function flagAccountForLiquidation(address account) external;

    // Restricted: used internally to Synthetix
    function removeAccountInLiquidation(address account) external;

    function checkAndRemoveAccountInLiquidation(address account) external;
}
```

IRewardEscrow.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/irewardescrow
interface IRewardEscrow {
    // Views
    function balanceOf(address account) external view returns (uint);

    function numVestingEntries(address account) external view returns (uint);

    function totalEscrowedAccountBalance(address account) external view returns (uint);
```

```
        function totalVestedAccountBalance(address account) external view returns (uint);

        // Mutative functions
        function appendVestingEntry(address account, uint quantity) external;

        function vest() external;
}
```

IRewardsDistribution.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/irewardsdistribution
interface IRewardsDistribution {
        // Structs
        struct DistributionData {
                address destination;
                uint amount;
        }

        // Views
        function authority() external view returns (address);

        function distributions(uint index) external view returns (address destination, uint amount); // DistributionData

        function distributionsLength() external view returns (uint);

        // Mutative Functions
        function distributeRewards(uint amount) external returns (bool);
}
```

IStakingRewards.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/istakingrewards
interface IStakingRewards {
        // Views
        function lastTimeRewardApplicable() external view returns (uint256);

        function rewardPerToken() external view returns (uint256);

        function earned(address account) external view returns (uint256);

        function getRewardForDuration() external view returns (uint256);

        function totalSupply() external view returns (uint256);

        function balanceOf(address account) external view returns (uint256);

        // Mutative

        function stake(uint256 amount) external;

        function withdraw(uint256 amount) external;

        function getReward() external;

        function exit() external;
}
```

ISupplySchedule.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/isupplyschedule
interface ISupplySchedule {
        // Views
        function mintableSupply() external view returns (uint);

        function isMintable() external view returns (bool);

        // Mutative functions
        function recordMintEvent(uint supplyMinted) external returns (bool);
}
```

ISynth.sol

```
pragma solidity >=0.4.24;
```

```
// https://docs.synthetix.io/contracts/source/interfaces/isynth
interface ISynth {
    // Views
    function currencyKey() external view returns (bytes32);

    function transferableSynths(address account) external view returns (uint);

    // Mutative functions
    function transferAndSettle(address to, uint value) external returns (bool);

    function transferFromAndSettle(
        address from,
        address to,
        uint value
    ) external returns (bool);

    // Restricted: used internally to Synthetix
    function burn(address account, uint amount) external;

    function issue(address account, uint amount) external;
}
```

ISynthetix.sol

```
pragma solidity >=0.4.24;

import "./ISynth.sol";
import "./IVirtualSynth.sol";


// https://docs.synthetix.io/contracts/source/interfaces/isynthetix
interface ISynthetix {
    // Views
    function anySynthOrSNXRateIsInvalid() external view returns (bool anyRateInvalid);

    function availableCurrencyKeys() external view returns (bytes32[] memory);

    function availableSynthCount() external view returns (uint);

    function availableSynths(uint index) external view returns (ISynth);

    function collateral(address account) external view returns (uint);

    function collateralisationRatio(address issuer) external view returns (uint);

    function debtBalanceOf(address issuer, bytes32 currencyKey) external view returns (uint);

    function isWaitingPeriod(bytes32 currencyKey) external view returns (bool);

    function maxIssuableSynths(address issuer) external view returns (uint maxIssuable);

    function remainingIssuableSynths(address issuer)
        external
        view
        returns (
            uint maxIssuable,
            uint alreadyIssued,
            uint totalSystemDebt
        );

    function synths(bytes32 currencyKey) external view returns (ISynth);

    function synthsByAddress(address synthAddress) external view returns (bytes32);

    function totalIssuedSynths(bytes32 currencyKey) external view returns (uint);

    function totalIssuedSynthsExcludeEtherCollateral(bytes32 currencyKey) external view returns (uint);

    function transferableSynthetix(address account) external view returns (uint transferable);

    // Mutative Functions
    function burnSynths(uint amount) external;

    function burnSynthsOnBehalf(address burnForAddress, uint amount) external;

    function burnSynthsToTarget() external;

    function burnSynthsToTargetOnBehalf(address burnForAddress) external;

    function exchange(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external returns (uint amountReceived);
```

```
    function exchangeOnBehalf(
        address exchangeForAddress,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external returns (uint amountReceived);

    function exchangeWithTracking(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external returns (uint amountReceived);

    function exchangeOnBehalfWithTracking(
        address exchangeForAddress,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external returns (uint amountReceived);

    function exchangeWithVirtual(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        bytes32 trackingCode
    ) external returns (uint amountReceived, IVirtualSynth vSynth);

    function issueMaxSynths() external;

    function issueMaxSynthsOnBehalf(address issueForAddress) external;

    function issueSynths(uint amount) external;

    function issueSynthsOnBehalf(address issueForAddress, uint amount) external;

    function mint() external returns (bool);

    function settle(bytes32 currencyKey)
        external
        returns (
            uint reclaimed,
            uint refunded,
            uint numEntries
        );

    function liquidateDelinquentAccount(address account, uint susdAmount) external returns (bool);

    // Restricted Functions

    function mintSecondary(address account, uint amount) external;

    function mintSecondaryRewards(uint amount) external;

    function burnSecondary(address account, uint amount) external;
}
```

ISynthetixBridgeToBase.sol

```
pragma solidity >=0.4.24;


interface ISynthetixBridgeToBase {
    // invoked by users on L2
    function initiateWithdrawal(uint amount) external;

    // invoked by the xDomain messenger on L2
    function mintSecondaryFromDeposit(address account, uint amount) external;

    // invoked by the xDomain messenger on L2
    function mintSecondaryFromDepositForRewards(uint amount) external;
}
```

ISynthetixBridgeToOptimism.sol

```
pragma solidity >=0.4.24;


interface ISynthetixBridgeToOptimism {
    // invoked by users on L1
    function deposit(uint amount) external;
```

```
        // invoked by the relayer on L1
        function completeWithdrawal(address account, uint amount) external;
}
```

ISynthetixState.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/isynthetixstate
interface ISynthetixState {
        // Views
        function debtLedger(uint index) external view returns (uint);

        function issuanceData(address account) external view returns (uint initialDebtOwnership, uint debtEntryIndex);

        function debtLedgerLength() external view returns (uint);

        function hasIssued(address account) external view returns (bool);

        function lastDebtLedgerEntry() external view returns (uint);

        // Mutative functions
        function incrementTotalIssuerCount() external;

        function decrementTotalIssuerCount() external;

        function setCurrentIssuanceData(address account, uint initialDebtOwnership) external;

        function appendDebtLedgerValue(uint value) external;

        function clearIssuanceData(address account) external;
}
```

ISystemSettings.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/isystemsettings
interface ISystemSettings {
        // Views
        function priceDeviationThresholdFactor() external view returns (uint);

        function waitingPeriodSecs() external view returns (uint);

        function issuanceRatio() external view returns (uint);

        function feePeriodDuration() external view returns (uint);

        function targetThreshold() external view returns (uint);

        function liquidationDelay() external view returns (uint);

        function liquidationRatio() external view returns (uint);

        function liquidationPenalty() external view returns (uint);

        function rateStalePeriod() external view returns (uint);

        function exchangeFeeRate(bytes32 currencyKey) external view returns (uint);

        function minimumStakeTime() external view returns (uint);
}
```

ISystemStatus.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/isystemstatus
interface ISystemStatus {
        struct Status {
                bool canSuspend;
                bool canResume;
        }

        struct Suspension {
                bool suspended;
                // reason is an integer code,
                // 0 => no reason, 1 => upgrading, 2+ => defined by system usage
                uint248 reason;
        }
```

```
    // Views
    function accessControl(bytes32 section, address account) external view returns (bool canSuspend, bool canResume);

    function requireSystemActive() external view;

    function requireIssuanceActive() external view;

    function requireExchangeActive() external view;

    function requireSynthActive(bytes32 currencyKey) external view;

    function requireSynthsActive(bytes32 sourceCurrencyKey, bytes32 destinationCurrencyKey) external view;

    function synthSuspension(bytes32 currencyKey) external view returns (bool suspended, uint248 reason);

    // Restricted functions
    function suspendSynth(bytes32 currencyKey, uint256 reason) external;

    function updateAccessControl(
        bytes32 section,
        address account,
        bool canSuspend,
        bool canResume
    ) external;
}
```

ITradingRewards.sol

```
pragma solidity >=0.4.24;


// https://docs.synthetix.io/contracts/source/interfaces/itradingrewards
interface ITradingRewards {
    /* ========== VIEWS ========== */

    function getAvailableRewards() external view returns (uint);

    function getUnassignedRewards() external view returns (uint);

    function getRewardsToken() external view returns (address);

    function getPeriodController() external view returns (address);

    function getCurrentPeriod() external view returns (uint);

    function getPeriodIsClaimable(uint periodID) external view returns (bool);

    function getPeriodIsFinalized(uint periodID) external view returns (bool);

    function getPeriodRecordedFees(uint periodID) external view returns (uint);

    function getPeriodTotalRewards(uint periodID) external view returns (uint);

    function getPeriodAvailableRewards(uint periodID) external view returns (uint);

    function getUnaccountedFeesForAccountForPeriod(address account, uint periodID) external view returns (uint);

    function getAvailableRewardsForAccountForPeriod(address account, uint periodID) external view returns (uint);

    function getAvailableRewardsForAccountForPeriods(address account, uint[] calldata periodIDs)
        external
        view
        returns (uint totalRewards);

    /* ========== MUTATIVE FUNCTIONS ========== */

    function claimRewardsForPeriod(uint periodID) external;

    function claimRewardsForPeriods(uint[] calldata periodIDs) external;

    /* ========== RESTRICTED FUNCTIONS ========== */

    function recordExchangeFeeForAccount(uint usdFeeAmount, address account) external;

    function closeCurrentPeriodWithRewards(uint rewards) external;

    function recoverTokens(address tokenAddress, address recoverAddress) external;

    function recoverUnassignedRewardTokens(address recoverAddress) external;

    function recoverAssignedRewardTokensAndDestroyPeriod(address recoverAddress, uint periodID) external;
```

```
        function setPeriodController(address newPeriodController) external;
}
```

IVirtualSynth.sol

```
pragma solidity >=0.4.24;

import "./ISynth.sol";


interface IVirtualSynth {
        // Views
        function balanceOfUnderlying(address account) external view returns (uint);

        function rate() external view returns (uint);

        function readyToSettle() external view returns (bool);

        function secsLeftInWaitingPeriod() external view returns (uint);

        function settled() external view returns (bool);

        function synth() external view returns (ISynth);

        // Mutative functions
        function settle(address account) external;
}
```

Issuer.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/IIssuer.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/ISynth.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/ISynthetixState.sol";
import "./interfaces/IExchanger.sol";
import "./interfaces/IDelegateApprovals.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/IEtherCollateral.sol";
import "./interfaces/IEtherCollateralsUSD.sol";
import "./interfaces/IRewardEscrow.sol";
import "./interfaces/IHasBalance.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/ILiquidations.sol";
import "./interfaces/IDebtCache.sol";


interface IIssuerInternalDebtCache {
        function updateCachedSynthDebtWithRate(bytes32 currencyKey, uint currencyRate) external;

        function updateCachedSynthDebtsWithRates(bytes32[] calldata currencyKeys, uint[] calldata currencyRates)
    external;

        function updateDebtCacheValidity(bool currentlyInvalid) external;

        function cacheInfo()
                external
                view
                returns (
                        uint cachedDebt,
                        uint timestamp,
                        bool isInvalid,
                        bool isStale
                );
}


// https://docs.synthetix.io/contracts/source/contracts/issuer
contract Issuer is Owned, MixinResolver, MixinSystemSettings, IIssuer {
        using SafeMath for uint;
        using SafeDecimalMath for uint;

        // Available Synths which can be used with the system
        ISynth[] public availableSynths;
```

```
    mapping(bytes32 => ISynth) public synths;
    mapping(address => bytes32) public synthsByAddress;

    /* ========== ENCODED NAMES ========== */

    bytes32 internal constant zUSD = "zUSD";
    bytes32 internal constant zBNB = "zBNB";
    bytes32 internal constant HZN = "HZN";

    // Flexible storage names

    bytes32 public constant CONTRACT_NAME = "Issuer";
    bytes32 internal constant LAST_ISSUE_EVENT = "lastIssueEvent";

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 private constant CONTRACT_SYNTHETIXSTATE = "SynthetixState";
    bytes32 private constant CONTRACT_FEEPOOL = "FeePool";
    bytes32 private constant CONTRACT_DELEGATEAPPROVALS = "DelegateApprovals";
    bytes32 private constant CONTRACT_ETHERCOLLATERAL = "EtherCollateral";
    bytes32 private constant CONTRACT_ETHERCOLLATERAL_SUSD = "EtherCollateralsUSD";
    bytes32 private constant CONTRACT_REWARDESCROW = "RewardEscrow";
    bytes32 private constant CONTRACT_SYNTHETIXESCROW = "SynthetixEscrow";
    bytes32 private constant CONTRACT_LIQUIDATIONS = "Liquidations";
    bytes32 private constant CONTRACT_FLEXIBLESTORAGE = "FlexibleStorage";
    bytes32 private constant CONTRACT_DEBTCACHE = "DebtCache";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYNTHETIX,
        CONTRACT_EXCHANGER,
        CONTRACT_EXRATES,
        CONTRACT_SYNTHETIXSTATE,
        CONTRACT_FEEPOOL,
        CONTRACT_DELEGATEAPPROVALS,
        CONTRACT_ETHERCOLLATERAL,
        CONTRACT_ETHERCOLLATERAL_SUSD,
        CONTRACT_REWARDESCROW,
        CONTRACT_SYNTHETIXESCROW,
        CONTRACT_LIQUIDATIONS,
        CONTRACT_FLEXIBLESTORAGE,
        CONTRACT_DEBTCACHE
    ];

    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {}

    /* ========== VIEWS ========== */

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Synthetix address"));
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    function synthetixState() internal view returns (ISynthetixState) {
        return ISynthetixState(requireAndGetAddress(CONTRACT_SYNTHETIXSTATE, "Missing SynthetixState address"));
    }

    function feePool() internal view returns (IFeePool) {
        return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL, "Missing FeePool address"));
    }

    function liquidations() internal view returns (ILiquidations) {
        return ILiquidations(requireAndGetAddress(CONTRACT_LIQUIDATIONS, "Missing Liquidations address"));
    }

    function delegateApprovals() internal view returns (IDelegateApprovals) {
        return IDelegateApprovals(requireAndGetAddress(CONTRACT_DELEGATEAPPROVALS, "Missing DelegateApprovals address"));
    }
```

```
    function etherCollateral() internal view returns (IEtherCollateral) {
        return IEtherCollateral(requireAndGetAddress(CONTRACT_ETHERCOLLATERAL, "Missing
EtherCollateral address"));
    }

    function etherCollateralsUSD() internal view returns (IEtherCollateralsUSD) {
        return
            IEtherCollateralsUSD(requireAndGetAddress(CONTRACT_ETHERCOLLATERAL_SUSD,
"Missing EtherCollateralsUSD address"));
    }

    function rewardEscrow() internal view returns (IRewardEscrow) {
        return IRewardEscrow(requireAndGetAddress(CONTRACT_REWARDESCROW, "Missing
RewardEscrow address"));
    }

    function synthetixEscrow() internal view returns (IHasBalance) {
        return IHasBalance(requireAndGetAddress(CONTRACT_SYNTHETIXESCROW, "Missing
SynthetixEscrow address"));
    }

    function debtCache() internal view returns (IIssuerInternalDebtCache) {
        return IIssuerInternalDebtCache(requireAndGetAddress(CONTRACT_DEBTCACHE, "Missing
DebtCache address"));
    }

    function issuanceRatio() external view returns (uint) {
        return getIssuanceRatio();
    }

    function _availableCurrencyKeysWithOptionalSNX(bool withHZN) internal view returns (bytes32[] memory)
{
        bytes32[] memory currencyKeys = new bytes32[](availableSynths.length + (withHZN ? 1 : 0));

        for (uint i = 0; i < availableSynths.length; i++) {
            currencyKeys[i] = synthsByAddress[address(availableSynths[i])];
        }

        if (withHZN) {
            currencyKeys[availableSynths.length] = HZN;
        }

        return currencyKeys;
    }

    function _totalIssuedSynths(bytes32 currencyKey, bool excludeEtherCollateral)
        internal
        view
        returns (uint totalIssued, bool anyRateIsInvalid)
    {
        (uint debt, , bool cacheIsInvalid, bool cacheIsStale) = debtCache().cacheInfo();
        anyRateIsInvalid = cacheIsInvalid || cacheIsStale;

        IExchangeRates exRates = exchangeRates();

        // Add total issued synths from Ether Collateral back into the total if not excluded
        if (!excludeEtherCollateral) {
            // Add ether collateral zUSD
            debt = debt.add(etherCollateralsUSD().totalIssuedSynths());

            // Add ether collateral zBNB
            (uint ethRate, bool ethRateInvalid) = exRates.rateAndInvalid(zBNB);
            uint ethIssuedDebt = etherCollateral().totalIssuedSynths().multiplyDecimalRound(ethRate);
            debt = debt.add(ethIssuedDebt);
            anyRateIsInvalid = anyRateIsInvalid || ethRateInvalid;
        }

        if (currencyKey == zUSD) {
            return (debt, anyRateIsInvalid);
        }

        (uint currencyRate, bool currencyRateInvalid) = exRates.rateAndInvalid(currencyKey);
        return (debt.divideDecimalRound(currencyRate), anyRateIsInvalid || currencyRateInvalid);
    }

    function _debtBalanceOfAndTotalDebt(address _issuer, bytes32 currencyKey)
        internal
        view
        returns (
            uint debtBalance,
            uint totalSystemValue,
            bool anyRateIsInvalid
        )
    {
        ISynthetixState state = synthetixState();

        // What was their initial debt ownership?
```

```
        (uint initialDebtOwnership, uint debtEntryIndex) = state.issuanceData(_issuer);

        // What's the total value of the system excluding ETH backed synths in their requested currency?
        (totalSystemValue, anyRateIsInvalid) = _totalIssuedSynths(currencyKey, true);

        // If it's zero, they haven't issued, and they have no debt.
        // Note: it's more gas intensive to put this check here rather than before _totalIssuedSynths
        // if they have 0 HZN, but it's a necessary trade-off
        if (initialDebtOwnership == 0) return (0, totalSystemValue, anyRateIsInvalid);

        // Figure out the global debt percentage delta from when they entered the system.
        // This is a high precision integer of 27 (1e27) decimals.
        uint currentDebtOwnership = state
            .lastDebtLedgerEntry()
            .divideDecimalRoundPrecise(state.debtLedger(debtEntryIndex))
            .multiplyDecimalRoundPrecise(initialDebtOwnership);

        // Their debt balance is their portion of the total system value.
        uint highPrecisionBalance =
totalSystemValue.decimalToPreciseDecimal().multiplyDecimalRoundPrecise(
            currentDebtOwnership
        );

        // Convert back into 18 decimals (1e18)
        debtBalance = highPrecisionBalance.preciseDecimalToDecimal();
    }

    function _canBurnSynths(address account) internal view returns (bool) {
        return now >= _lastIssueEvent(account).add(getMinimumStakeTime());
    }

    function _lastIssueEvent(address account) internal view returns (uint) {
        //   Get the timestamp of the last issue this account made
        return flexibleStorage().getUIntValue(CONTRACT_NAME,
keccak256(abi.encodePacked(LAST_ISSUE_EVENT, account)));
    }

    function _remainingIssuableSynths(address _issuer)
        internal
        view
        returns (
            uint maxIssuable,
            uint alreadyIssued,
            uint totalSystemDebt,
            bool anyRateIsInvalid
        )
    {
        (alreadyIssued, totalSystemDebt, anyRateIsInvalid) = _debtBalanceOfAndTotalDebt(_issuer, zUSD);
        (uint issuable, bool isInvalid) = _maxIssuableSynths(_issuer);
        maxIssuable = issuable;
        anyRateIsInvalid = anyRateIsInvalid || isInvalid;

        if (alreadyIssued >= maxIssuable) {
            maxIssuable = 0;
        } else {
            maxIssuable = maxIssuable.sub(alreadyIssued);
        }
    }

    function _hznToUSD(uint amount, uint hznRate) internal pure returns (uint) {
        return amount.multiplyDecimalRound(hznRate);
    }

    function _usdToHZN(uint amount, uint hznRate) internal pure returns (uint) {
        return amount.divideDecimalRound(hznRate);
    }

    function _maxIssuableSynths(address _issuer) internal view returns (uint, bool) {
        // What is the value of their HZN balance in zUSD
        (uint hznRate, bool isInvalid) = exchangeRates().rateAndInvalid(HZN);
        uint destinationValue = _hznToUSD(_collateral(_issuer), hznRate);

        // They're allowed to issue up to issuanceRatio of that value
        return (destinationValue.multiplyDecimal(getIssuanceRatio()), isInvalid);
    }

    function _collateralisationRatio(address _issuer) internal view returns (uint, bool) {
        uint totalOwnedSynthetix = _collateral(_issuer);

        (uint debtBalance, , bool anyRateIsInvalid) = _debtBalanceOfAndTotalDebt(_issuer, HZN);

        // it's more gas intensive to put this check here if they have 0 HZN, but it complies with the interface
        if (totalOwnedSynthetix == 0) return (0, anyRateIsInvalid);

        return (debtBalance.divideDecimalRound(totalOwnedSynthetix), anyRateIsInvalid);
    }
```

```
    function _collateral(address account) internal view returns (uint) {
        uint balance = IERC20(address(synthetix())).balanceOf(account);

        if (address(synthetixEscrow()) != address(0)) {
            balance = balance.add(synthetixEscrow().balanceOf(account));
        }

        if (address(rewardEscrow()) != address(0)) {
            balance = balance.add(rewardEscrow().balanceOf(account));
        }

        return balance;
    }

    function minimumStakeTime() external view returns (uint) {
        return getMinimumStakeTime();
    }

    function canBurnSynths(address account) external view returns (bool) {
        return _canBurnSynths(account);
    }

    function availableCurrencyKeys() external view returns (bytes32[] memory) {
        return _availableCurrencyKeysWithOptionalSNX(false);
    }

    function availableSynthCount() external view returns (uint) {
        return availableSynths.length;
    }

    function anySynthOrSNXRateIsInvalid() external view returns (bool anyRateInvalid) {
        (, anyRateInvalid) = exchangeRates().ratesAndInvalidForCurrencies(_availableCurrencyKeysWithOptionalSNX(true));
    }

    function totalIssuedSynths(bytes32 currencyKey, bool excludeEtherCollateral) external view returns (uint totalIssued) {
        (totalIssued, ) = _totalIssuedSynths(currencyKey, excludeEtherCollateral);
    }

    function lastIssueEvent(address account) external view returns (uint) {
        return _lastIssueEvent(account);
    }

    function collateralisationRatio(address _issuer) external view returns (uint cratio) {
        (cratio, ) = _collateralisationRatio(_issuer);
    }

    function collateralisationRatioAndAnyRatesInvalid(address _issuer)
        external
        view
        returns (uint cratio, bool anyRateIsInvalid)
    {
        return _collateralisationRatio(_issuer);
    }

    function collateral(address account) external view returns (uint) {
        return _collateral(account);
    }

    function debtBalanceOf(address _issuer, bytes32 currencyKey) external view returns (uint debtBalance) {
        ISynthetixState state = synthetixState();

        // What was their initial debt ownership?
        (uint initialDebtOwnership, ) = state.issuanceData(_issuer);

        // If it's zero, they haven't issued, and they have no debt.
        if (initialDebtOwnership == 0) return 0;

        (debtBalance, , ) = _debtBalanceOfAndTotalDebt(_issuer, currencyKey);
    }

    function remainingIssuableSynths(address _issuer)
        external
        view
        returns (
            uint maxIssuable,
            uint alreadyIssued,
            uint totalSystemDebt
        )
    {
        (maxIssuable, alreadyIssued, totalSystemDebt, ) = _remainingIssuableSynths(_issuer);
    }

    function maxIssuableSynths(address _issuer) external view returns (uint) {
        (uint maxIssuable, ) = _maxIssuableSynths(_issuer);
        return maxIssuable;
```

```
}

function transferableSynthetixAndAnyRateIsInvalid(address account, uint balance)
        external
        view
        returns (uint transferable, bool anyRateIsInvalid)
{
        // How many HZN do they have, excluding escrow?
        // Note: We're excluding escrow here because we're interested in their transferable amount
        // and escrowed HZN are not transferable.

        // How many of those will be locked by the amount they've issued?
        // Assuming issuance ratio is 20%, then issuing 20 HZN of value would require
        // 100 HZN to be locked in their wallet to maintain their collateralisation ratio
        // The locked synthetix value can exceed their balance.
        uint debtBalance;
        (debtBalance, , anyRateIsInvalid) = _debtBalanceOfAndTotalDebt(account, HZN);
        uint lockedSynthetixValue = debtBalance.divideDecimalRound(getIssuanceRatio());

        // If we exceed the balance, no HZN are transferable, otherwise the difference is.
        if (lockedSynthetixValue >= balance) {
                transferable = 0;
        } else {
                transferable = balance.sub(lockedSynthetixValue);
        }
}

function getSynths(bytes32[] calldata currencyKeys) external view returns (ISynth[] memory) {
        uint numKeys = currencyKeys.length;
        ISynth[] memory addresses = new ISynth[](numKeys);

        for (uint i = 0; i < numKeys; i++) {
                addresses[i] = synths[currencyKeys[i]];
        }

        return addresses;
}

/* ========== MUTATIVE FUNCTIONS ========== */

function _addSynth(ISynth synth) internal {
        bytes32 currencyKey = synth.currencyKey();
        require(synths[currencyKey] == ISynth(0), "Zasset exists");
        require(synthsByAddress[address(synth)] == bytes32(0), "Zasset address already exists");

        availableSynths.push(synth);
        synths[currencyKey] = synth;
        synthsByAddress[address(synth)] = currencyKey;

        emit SynthAdded(currencyKey, address(synth));
}

function addSynth(ISynth synth) external onlyOwner {
        _addSynth(synth);
        // Invalidate the cache to force a snapshot to be recomputed. If a synth were to be added
        // back to the system and it still somehow had cached debt, this would force the value to be
        // updated.
        debtCache().updateDebtCacheValidity(true);
}

function addSynths(ISynth[] calldata synthsToAdd) external onlyOwner {
        uint numSynths = synthsToAdd.length;
        for (uint i = 0; i < numSynths; i++) {
                _addSynth(synthsToAdd[i]);
        }

        // Invalidate the cache to force a snapshot to be recomputed.
        debtCache().updateDebtCacheValidity(true);
}

function _removeSynth(bytes32 currencyKey) internal {
        address synthToRemove = address(synths[currencyKey]);
        require(synthToRemove != address(0), "Zasset does not exist");
        require(IERC20(synthToRemove).totalSupply() == 0, "Zasset supply exists");
        require(currencyKey != zUSD, "Cannot remove zasset");

        // Remove the synth from the availableSynths array.
        for (uint i = 0; i < availableSynths.length; i++) {
                if (address(availableSynths[i]) == synthToRemove) {
                        delete availableSynths[i];

                        // Copy the last synth into the place of the one we just deleted
                        // If there's only one synth, this is synths[0] = synths[0].
                        // If we're deleting the last one, it's also a NOOP in the same way.
                        availableSynths[i] = availableSynths[availableSynths.length - 1];

                        // Decrease the size of the array by one.
```

```
                availableSynths.length--;

                break;
            }
        }

        // And remove it from the synths mapping
        delete synthsByAddress[synthToRemove];
        delete synths[currencyKey];

        emit SynthRemoved(currencyKey, synthToRemove);
    }

    function removeSynth(bytes32 currencyKey) external onlyOwner {
        // Remove its contribution from the debt pool snapshot, and
        // invalidate the cache to force a new snapshot.
        IIssuerInternalDebtCache cache = debtCache();
        cache.updateCachedSynthDebtWithRate(currencyKey, 0);
        cache.updateDebtCacheValidity(true);

        _removeSynth(currencyKey);
    }

    function removeSynths(bytes32[] calldata currencyKeys) external onlyOwner {
        uint numKeys = currencyKeys.length;

        // Remove their contributions from the debt pool snapshot, and
        // invalidate the cache to force a new snapshot.
        IIssuerInternalDebtCache cache = debtCache();
        uint[] memory zeroRates = new uint[](numKeys);
        cache.updateCachedSynthDebtsWithRates(currencyKeys, zeroRates);
        cache.updateDebtCacheValidity(true);

        for (uint i = 0; i < numKeys; i++) {
            _removeSynth(currencyKeys[i]);
        }
    }

    function issueSynths(address from, uint amount) external onlySynthetix {
        _issueSynths(from, amount, false);
    }

    function issueMaxSynths(address from) external onlySynthetix {
        _issueSynths(from, 0, true);
    }

    function issueSynthsOnBehalf(
        address issueForAddress,
        address from,
        uint amount
    ) external onlySynthetix {
        _requireCanIssueOnBehalf(issueForAddress, from);
        _issueSynths(issueForAddress, amount, false);
    }

    function issueMaxSynthsOnBehalf(address issueForAddress, address from) external onlySynthetix {
        _requireCanIssueOnBehalf(issueForAddress, from);
        _issueSynths(issueForAddress, 0, true);
    }

    function burnSynths(address from, uint amount) external onlySynthetix {
        _voluntaryBurnSynths(from, amount, false);
    }

    function burnSynthsOnBehalf(
        address burnForAddress,
        address from,
        uint amount
    ) external onlySynthetix {
        _requireCanBurnOnBehalf(burnForAddress, from);
        _voluntaryBurnSynths(burnForAddress, amount, false);
    }

    function burnSynthsToTarget(address from) external onlySynthetix {
        _voluntaryBurnSynths(from, 0, true);
    }

    function burnSynthsToTargetOnBehalf(address burnForAddress, address from) external onlySynthetix {
        _requireCanBurnOnBehalf(burnForAddress, from);
        _voluntaryBurnSynths(burnForAddress, 0, true);
    }

    function liquidateDelinquentAccount(
        address account,
        uint zUSDAmount,
        address liquidator
    ) external onlySynthetix returns (uint totalRedeemed, uint amountToLiquidate) {
```

```
        // Ensure waitingPeriod and zUSD balance is settled as burning impacts the size of debt pool
        require(!exchanger().hasWaitingPeriodOrSettlementOwing(liquidator, zUSD), "zUSD needs to be settled");

        // Check account is liquidation open
        require(liquidations().isOpenForLiquidation(account), "Account not open for liquidation");

        // require liquidator has enough zUSD
        require(IERC20(address(synths[zUSD])).balanceOf(liquidator) >= zUSDAmount, "Not enough zUSD");

        uint liquidationPenalty = liquidations().liquidationPenalty();

        // What is their debt in zUSD?
        (uint debtBalance, uint totalDebtIssued, bool anyRateIsInvalid) = _debtBalanceOfAndTotalDebt(account, zUSD);
        (uint hznRate, bool snxRateInvalid) = exchangeRates().rateAndInvalid(HZN);
        _requireRatesNotInvalid(anyRateIsInvalid || snxRateInvalid);

        uint collateralForAccount = _collateral(account);
        uint amountToFixRatio = liquidations().calculateAmountToFixCollateral(
            debtBalance,
            _hznToUSD(collateralForAccount, hznRate)
        );

        // Cap amount to liquidate to repair collateral ratio based on issuance ratio
        amountToLiquidate = amountToFixRatio < zUSDAmount ? amountToFixRatio : zUSDAmount;

        // what's the equivalent amount of HZN for the amountToLiquidate?
        uint hznRedeemed = _usdToHZN(amountToLiquidate, hznRate);

        // Add penalty
        totalRedeemed = hznRedeemed.multiplyDecimal(SafeDecimalMath.unit().add(liquidationPenalty));

        // if total HZN to redeem is greater than account's collateral
        // account is under collateralised, liquidate all collateral and reduce zUSD to burn
        // an insurance fund will be added to cover these undercollateralised positions
        if (totalRedeemed > collateralForAccount) {
            // set totalRedeemed to all collateral
            totalRedeemed = collateralForAccount;

            // whats the equivalent zUSD to burn for all collateral less penalty
            amountToLiquidate = _hznToUSD(
                collateralForAccount.divideDecimal(SafeDecimalMath.unit().add(liquidationPenalty)),
                hznRate
            );
        }

        // burn zUSD from messageSender (liquidator) and reduce account's debt
        _burnSynths(account, liquidator, amountToLiquidate, debtBalance, totalDebtIssued);

        if (amountToLiquidate == amountToFixRatio) {
            // Remove liquidation
            liquidations().removeAccountInLiquidation(account);
        }
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _requireRatesNotInvalid(bool anyRateIsInvalid) internal pure {
        require(!anyRateIsInvalid, "A zasset or HZN rate is invalid");
    }

    function _requireCanIssueOnBehalf(address issueForAddress, address from) internal view {
        require(delegateApprovals().canIssueFor(issueForAddress, from), "Not approved to act on behalf");
    }

    function _requireCanBurnOnBehalf(address burnForAddress, address from) internal view {
        require(delegateApprovals().canBurnFor(burnForAddress, from), "Not approved to act on behalf");
    }

    function _issueSynths(
        address from,
        uint amount,
        bool issueMax
    ) internal {
        (uint maxIssuable, uint existingDebt, uint totalSystemDebt, bool anyRateIsInvalid) = _remainingIssuableSynths(from);
        _requireRatesNotInvalid(anyRateIsInvalid);

        if (!issueMax) {
            require(amount <= maxIssuable, "Amount too large");
        } else {
            amount = maxIssuable;
        }

        // Keep track of the debt they're about to create
        _addToDebtRegister(from, amount, existingDebt, totalSystemDebt);
```

```
        // record issue timestamp
        _setLastIssueEvent(from);

        // Create their synths
        synths[zUSD].issue(from, amount);

        // Account for the issued debt in the cache
        debtCache().updateCachedSynthDebtWithRate(zUSD, SafeDecimalMath.unit());

        // Store their locked HZN amount to determine their fee % for the period
        _appendAccountIssuanceRecord(from);
    }

    function _burnSynths(
        address debtAccount,
        address burnAccount,
        uint amount,
        uint existingDebt,
        uint totalDebtIssued
    ) internal returns (uint amountBurnt) {
        // liquidation requires zUSD to be already settled / not in waiting period

        // If they're trying to burn more debt than they actually owe, rather than fail the transaction, let's just
        // clear their debt and leave them be.
        amountBurnt = existingDebt < amount ? existingDebt : amount;

        // Remove liquidated debt from the ledger
        _removeFromDebtRegister(debtAccount, amountBurnt, existingDebt, totalDebtIssued);

        // synth.burn does a safe subtraction on balance (so it will revert if there are not enough synths).
        synths[zUSD].burn(burnAccount, amountBurnt);

        // Account for the burnt debt in the cache.
        debtCache().updateCachedSynthDebtWithRate(zUSD, SafeDecimalMath.unit());

        // Store their debtRatio against a fee period to determine their fee/rewards % for the period
        _appendAccountIssuanceRecord(debtAccount);
    }

    // If burning to target, `amount` is ignored, and the correct quantity of zUSD is burnt to reach the target
    // c-ratio, allowing fees to be claimed. In this case, pending settlements will be skipped as the user
    // will still have debt remaining after reaching their target.
    function _voluntaryBurnSynths(
        address from,
        uint amount,
        bool burnToTarget
    ) internal {
        if (!burnToTarget) {
            // If not burning to target, then burning requires that the minimum stake time has elapsed.
            require(_canBurnSynths(from), "Minimum stake time not reached");
            // First settle anything pending into zUSD as burning or issuing impacts the size of the debt pool
            (, uint refunded, uint numEntriesSettled) = exchanger().settle(from, zUSD);
            if (numEntriesSettled > 0) {
                amount = exchanger().calculateAmountAfterSettlement(from, zUSD, amount, refunded);
            }
        }

        (uint existingDebt, uint totalSystemValue, bool anyRateIsInvalid) = _debtBalanceOfAndTotalDebt(from,
zUSD);
        (uint maxIssuableSynthsForAccount, bool snxRateInvalid) = _maxIssuableSynths(from);
        _requireRatesNotInvalid(anyRateIsInvalid || snxRateInvalid);
        require(existingDebt > 0, "No debt to forgive");

        if (burnToTarget) {
            amount = existingDebt.sub(maxIssuableSynthsForAccount);
        }

        uint amountBurnt = _burnSynths(from, from, amount, existingDebt, totalSystemValue);

        // Check and remove liquidation if existingDebt after burning is <= maxIssuableSynths
        // Issuance ratio is fixed so should remove any liquidations
        if (existingDebt.sub(amountBurnt) <= maxIssuableSynthsForAccount) {
            liquidations().removeAccountInLiquidation(from);
        }
    }

    function _setLastIssueEvent(address account) internal {
        // Set the timestamp of the last issueSynths
        flexibleStorage().setUIntValue(
            CONTRACT_NAME,
            keccak256(abi.encodePacked(LAST_ISSUE_EVENT, account)),
            block.timestamp
        );
    }

    function _appendAccountIssuanceRecord(address from) internal {
```

```
        uint initialDebtOwnership;
        uint debtEntryIndex;
        (initialDebtOwnership, debtEntryIndex) = synthetixState().issuanceData(from);
        feePool().appendAccountIssuanceRecord(from, initialDebtOwnership, debtEntryIndex);
    }

    function _addToDebtRegister(
        address from,
        uint amount,
        uint existingDebt,
        uint totalDebtIssued
    ) internal {
        ISynthetixState state = synthetixState();

        // What will the new total be including the new value?
        uint newTotalDebtIssued = amount.add(totalDebtIssued);

        // What is their percentage (as a high precision int) of the total debt?
        uint debtPercentage = amount.divideDecimalRoundPrecise(newTotalDebtIssued);

        // And what effect does this percentage change have on the global debt holding of other issuers?
        // The delta specifically needs to not take into account any existing debt as it's already
        // accounted for in the delta from when they issued previously.
        // The delta is a high precision integer.
        uint delta = SafeDecimalMath.preciseUnit().sub(debtPercentage);

        // And what does their debt ownership look like including this previous stake?
        if (existingDebt > 0) {
            debtPercentage = amount.add(existingDebt).divideDecimalRoundPrecise(newTotalDebtIssued);
        } else {
            // If they have no debt, they're a new issuer; record this.
            state.incrementTotalIssuerCount();
        }

        // Save the debt entry parameters
        state.setCurrentIssuanceData(from, debtPercentage);

        // And if we're the first, push 1 as there was no effect to any other holders, otherwise push
        // the change for the rest of the debt holders. The debt ledger holds high precision integers.
        if (state.debtLedgerLength() > 0) {
            state.appendDebtLedgerValue(state.lastDebtLedgerEntry().multiplyDecimalRoundPrecise(delta));
        } else {
            state.appendDebtLedgerValue(SafeDecimalMath.preciseUnit());
        }
    }

    function _removeFromDebtRegister(
        address from,
        uint debtToRemove,
        uint existingDebt,
        uint totalDebtIssued
    ) internal {
        ISynthetixState state = synthetixState();

        // What will the new total after taking out the withdrawn amount
        uint newTotalDebtIssued = totalDebtIssued.sub(debtToRemove);

        uint delta = 0;

        // What will the debt delta be if there is any debt left?
        // Set delta to 0 if no more debt left in system after user
        if (newTotalDebtIssued > 0) {
            // What is the percentage of the withdrawn debt (as a high precision int) of the total debt after?
            uint debtPercentage = debtToRemove.divideDecimalRoundPrecise(newTotalDebtIssued);

            // And what effect does this percentage change have on the global debt holding of other issuers?
            // The delta specifically needs to not take into account any existing debt as it's already
            // accounted for in the delta from when they issued previously.
            delta = SafeDecimalMath.preciseUnit().add(debtPercentage);
        }

        // Are they exiting the system, or are they just decreasing their debt position?
        if (debtToRemove == existingDebt) {
            state.setCurrentIssuanceData(from, 0);
            state.decrementTotalIssuerCount();
        } else {
            // What percentage of the debt will they be left with?
            uint newDebt = existingDebt.sub(debtToRemove);
            uint newDebtPercentage = newDebt.divideDecimalRoundPrecise(newTotalDebtIssued);

            // Store the debt percentage and debt ledger as high precision integers
            state.setCurrentIssuanceData(from, newDebtPercentage);
        }

        // Update our cumulative ledger. This is also a high precision integer.
        state.appendDebtLedgerValue(state.lastDebtLedgerEntry().multiplyDecimalRoundPrecise(delta));
    }
```

```
    /* ========== MODIFIERS ========== */

    function _onlySynthetix() internal view {
        require(msg.sender == address(synthetix()), "Issuer: Only the synthetix contract can perform this action");
    }

    modifier onlySynthetix() {
        _onlySynthetix(); // Use an internal function to save code size.
        _;
    }

    /* ========== EVENTS ========== */

    event SynthAdded(bytes32 currencyKey, address synth);
    event SynthRemoved(bytes32 currencyKey, address synth);
}
```

LimitedSetup.sol

```
pragma solidity ^0.5.16;


// https://docs.synthetix.io/contracts/source/contracts/limitedsetup
contract LimitedSetup {
    uint public setupExpiryTime;

    /**
     * @dev LimitedSetup Constructor.
     * @param setupDuration The time the setup period will last for.
     */
    constructor(uint setupDuration) internal {
        setupExpiryTime = now + setupDuration;
    }

    modifier onlyDuringSetup {
        require(now < setupExpiryTime, "Can only perform this action during setup");
        _;
    }
}
```

Liquidations.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/ILiquidations.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./EternalStorage.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/IIssuer.sol";
import "./interfaces/ISystemStatus.sol";


// https://docs.synthetix.io/contracts/source/contracts/liquidations
contract Liquidations is Owned, MixinResolver, MixinSystemSettings, ILiquidations {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    struct LiquidationEntry {
        uint deadline;
        address caller;
    }

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_ETERNALSTORAGE_LIQUIDATIONS = "EternalStorageLiquidations";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";

    bytes32[24] private addressesToCache = [
        CONTRACT_SYSTEMSTATUS,
        CONTRACT_SYNTHETIX,
```

```
        CONTRACT_ETERNALSTORAGE_LIQUIDATIONS,
        CONTRACT_ISSUER,
        CONTRACT_EXRATES
    ];

    /* ========== CONSTANTS ========== */

    // Storage keys
    bytes32 public constant LIQUIDATION_DEADLINE = "LiquidationDeadline";
    bytes32 public constant LIQUIDATION_CALLER = "LiquidationCaller";

    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {}

    /* ========== VIEWS ========== */
    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    // refactor to synthetix storage eternal storage contract once that's ready
    function eternalStorageLiquidations() internal view returns (EternalStorage) {
        return
            EternalStorage(
                requireAndGetAddress(CONTRACT_ETERNALSTORAGE_LIQUIDATIONS, "Missing EternalStorageLiquidations address")
            );
    }

    function issuanceRatio() external view returns (uint) {
        return getIssuanceRatio();
    }

    function liquidationDelay() external view returns (uint) {
        return getLiquidationDelay();
    }

    function liquidationRatio() external view returns (uint) {
        return getLiquidationRatio();
    }

    function liquidationPenalty() external view returns (uint) {
        return getLiquidationPenalty();
    }

    function liquidationCollateralRatio() external view returns (uint) {
        return SafeDecimalMath.unit().divideDecimalRound(getLiquidationRatio());
    }

    function getLiquidationDeadlineForAccount(address account) external view returns (uint) {
        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);
        return liquidation.deadline;
    }

    function isOpenForLiquidation(address account) external view returns (bool) {
        uint accountCollateralisationRatio = synthetix().collateralisationRatio(account);

        // Liquidation closed if collateral ratio less than or equal target issuance Ratio
        // Account with no HZN collateral will also not be open for liquidation (ratio is 0)
        if (accountCollateralisationRatio <= getIssuanceRatio()) {
            return false;
        }

        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);

        // liquidation cap at issuanceRatio is checked above
        if (_deadlinePassed(liquidation.deadline)) {
            return true;
        }
        return false;
```

```
    }

    function isLiquidationDeadlinePassed(address account) external view returns (bool) {
        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);
        return _deadlinePassed(liquidation.deadline);
    }

    function _deadlinePassed(uint deadline) internal view returns (bool) {
        // check deadline is set > 0
        // check now > deadline
        return deadline > 0 && now > deadline;
    }

    /**
     * r = target issuance ratio
     * D = debt balance
     * V = Collateral
     * P = liquidation penalty
     * Calculates amount of synths = (D - V * r) / (1 - (1 + P) * r)
     */
    function calculateAmountToFixCollateral(uint debtBalance, uint collateral) external view returns (uint) {
        uint ratio = getIssuanceRatio();
        uint unit = SafeDecimalMath.unit();

        uint dividend = debtBalance.sub(collateral.multiplyDecimal(ratio));
        uint divisor = unit.sub(unit.add(getLiquidationPenalty()).multiplyDecimal(ratio));

        return dividend.divideDecimal(divisor);
    }

    // get liquidationEntry for account
    // returns deadline = 0 when not set
    function _getLiquidationEntryForAccount(address account) internal view returns (LiquidationEntry memory
_liquidation) {
        _liquidation.deadline                                                                                =
eternalStorageLiquidations().getUIntValue(_getKey(LIQUIDATION_DEADLINE, account));

        // liquidation caller not used
        _liquidation.caller = address(0);
    }

    function _getKey(bytes32 _scope, address _account) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked(_scope, _account));
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    // totalIssuedSynths checks synths for staleness
    // check HZN rate is not stale
    function flagAccountForLiquidation(address account) external rateNotInvalid("HZN") {
        systemStatus().requireSystemActive();

        require(getLiquidationRatio() > 0, "Liquidation ratio not set");
        require(getLiquidationDelay() > 0, "Liquidation delay not set");

        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);
        require(liquidation.deadline == 0, "Account already flagged for liquidation");

        uint accountsCollateralisationRatio = synthetix().collateralisationRatio(account);

        // if accounts issuance ratio is greater than or equal to liquidation ratio set liquidation entry
        require(
            accountsCollateralisationRatio >= getLiquidationRatio(),
            "Account issuance ratio is less than liquidation ratio"
        );

        uint deadline = now.add(getLiquidationDelay());

        _storeLiquidationEntry(account, deadline, msg.sender);

        emit AccountFlaggedForLiquidation(account, deadline);
    }

    // Internal function to remove account from liquidations
    // Does not check collateral ratio is fixed
    function removeAccountInLiquidation(address account) external onlyIssuer {
        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);
        if (liquidation.deadline > 0) {
            _removeLiquidationEntry(account);
        }
    }

    // Public function to allow an account to remove from liquidations
    // Checks collateral ratio is fixed - below target issuance ratio
    // Check HZN rate is not stale
    function checkAndRemoveAccountInLiquidation(address account) external rateNotInvalid("HZN") {
        systemStatus().requireSystemActive();
```

```
        LiquidationEntry memory liquidation = _getLiquidationEntryForAccount(account);

        require(liquidation.deadline > 0, "Account has no liquidation set");

        uint accountsCollateralisationRatio = synthetix().collateralisationRatio(account);

        // Remove from liquidations if accountsCollateralisationRatio is fixed (less than equal target issuance ratio)
        if (accountsCollateralisationRatio <= getIssuanceRatio()) {
            _removeLiquidationEntry(account);
        }
    }

    function _storeLiquidationEntry(
        address _account,
        uint _deadline,
        address _caller
    ) internal {
        // record liquidation deadline
        eternalStorageLiquidations().setUIntValue(_getKey(LIQUIDATION_DEADLINE, _account), _deadline);
        eternalStorageLiquidations().setAddressValue(_getKey(LIQUIDATION_CALLER, _account), _caller);
    }

    function _removeLiquidationEntry(address _account) internal {
        // delete liquidation deadline
        eternalStorageLiquidations().deleteUIntValue(_getKey(LIQUIDATION_DEADLINE, _account));
        // delete liquidation caller
        eternalStorageLiquidations().deleteAddressValue(_getKey(LIQUIDATION_CALLER, _account));

        emit AccountRemovedFromLiquidation(_account, now);
    }

    /* ========== MODIFIERS ========== */
    modifier onlyIssuer() {
        require(msg.sender == address(issuer()), "Liquidations: Only the Issuer contract can perform this action");
        _;
    }

    modifier rateNotInvalid(bytes32 currencyKey) {
        require(!exchangeRates().rateIsInvalid(currencyKey), "Rate invalid or not a zasset");
        _;
    }

    /* ========== EVENTS ========== */

    event AccountFlaggedForLiquidation(address indexed account, uint deadline);
    event AccountRemovedFromLiquidation(address indexed account, uint time);
}
```

Math.sol

```
pragma solidity ^0.5.16;

// Libraries
import "./SafeDecimalMath.sol";


// https://docs.synthetix.io/contracts/source/libraries/math
library Math {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    /**
     * @dev Uses "exponentiation by squaring" algorithm where cost is 0(logN)
     * vs 0(N) for naive repeated multiplication.
     * Calculates x^n with x as fixed-point and n as regular unsigned int.
     * Calculates to 18 digits of precision with SafeDecimalMath.unit()
     */
    function powDecimal(uint x, uint n) internal pure returns (uint) {
        // https://mpark.github.io/programming/2014/08/18/exponentiation-by-squaring/

        uint result = SafeDecimalMath.unit();
        while (n > 0) {
            if (n % 2 != 0) {
                result = result.multiplyDecimal(x);
            }
            x = x.multiplyDecimal(x);
            n /= 2;
        }
        return result;
    }
}
```

MintableSynthetix.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./BaseSynthetix.sol";


// https://docs.synthetix.io/contracts/source/contracts/mintablesynthetix
contract MintableSynthetix is BaseSynthetix {
    bytes32 private constant CONTRACT_SYNTHETIX_BRIDGE = "SynthetixBridgeToBase";

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        address _owner,
        uint _totalSupply,
        address _resolver
    ) public BaseSynthetix(_proxy, _tokenState, _owner, _totalSupply, _resolver) {
        appendToAddressCache(CONTRACT_SYNTHETIX_BRIDGE);
    }

    /* ========== INTERNALS ========== */

    function _mintSecondary(address account, uint amount) internal {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).add(amount));
        emitTransfer(address(this), account, amount);
        totalSupply = totalSupply.add(amount);
    }

    function onlyAllowFromBridge() internal view {
        require(msg.sender == synthetixBridge(), "Can only be invoked by the SynthetixBridgeToBase contract");
    }

    /* ========== MODIFIERS ========== */

    modifier onlyBridge() {
        onlyAllowFromBridge();
        _;
    }

    /* ========== VIEWS ========== */

    function synthetixBridge() internal view returns (address) {
        return requireAndGetAddress(CONTRACT_SYNTHETIX_BRIDGE, "Resolver is missing SynthetixBridgeToBase address");
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function mintSecondary(address account, uint amount) external onlyBridge {
        _mintSecondary(account, amount);
    }

    function mintSecondaryRewards(uint amount) external onlyBridge {
        IRewardsDistribution _rewardsDistribution = rewardsDistribution();
        _mintSecondary(address(_rewardsDistribution), amount);
        _rewardsDistribution.distributeRewards(amount);
    }

    function burnSecondary(address account, uint amount) external onlyBridge {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).sub(amount));
        emitTransfer(account, address(0), amount);
        totalSupply = totalSupply.sub(amount);
    }
}
```

MixinResolver.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";

// Internal references
import "./AddressResolver.sol";


// https://docs.synthetix.io/contracts/source/contracts/mixinresolver
contract MixinResolver is Owned {
    AddressResolver public resolver;

    mapping(bytes32 => address) private addressCache;

    bytes32[] public resolverAddressesRequired;
```

```
    uint public constant MAX_ADDRESSES_FROM_RESOLVER = 24;

    constructor(address _resolver, bytes32[MAX_ADDRESSES_FROM_RESOLVER] memory
_addressesToCache) internal {
        // This contract is abstract, and thus cannot be instantiated directly
        require(owner != address(0), "Owner must be set");

        for (uint i = 0; i < _addressesToCache.length; i++) {
            if (_addressesToCache[i] != bytes32(0)) {
                resolverAddressesRequired.push(_addressesToCache[i]);
            } else {
                // End early once an empty item is found - assumes there are no empty slots in
                // _addressesToCache
                break;
            }
        }
        resolver = AddressResolver(_resolver);
        // Do not sync the cache as addresses may not be in the resolver yet
    }

    /* ========== SETTERS ========== */
    function setResolverAndSyncCache(AddressResolver _resolver) external onlyOwner {
        resolver = _resolver;

        for (uint i = 0; i < resolverAddressesRequired.length; i++) {
            bytes32 name = resolverAddressesRequired[i];
            // Note: can only be invoked once the resolver has all the targets needed added
            addressCache[name] = resolver.requireAndGetAddress(name, "Resolver missing target");
        }
    }

    /* ========== VIEWS ========== */

    function requireAndGetAddress(bytes32 name, string memory reason) internal view returns (address) {
        address _foundAddress = addressCache[name];
        require(_foundAddress != address(0), reason);
        return _foundAddress;
    }

    // Note: this could be made external in a utility contract if addressCache was made public
    // (used for deployment)
    function isResolverCached(AddressResolver _resolver) external view returns (bool) {
        if (resolver != _resolver) {
            return false;
        }

        // otherwise, check everything
        for (uint i = 0; i < resolverAddressesRequired.length; i++) {
            bytes32 name = resolverAddressesRequired[i];
            // false if our cache is invalid or if the resolver doesn't have the required address
            if (resolver.getAddress(name) != addressCache[name] || addressCache[name] == address(0)) {
                return false;
            }
        }

        return true;
    }

    // Note: can be made external into a utility contract (used for deployment)
    function getResolverAddressesRequired()
        external
        view
        returns (bytes32[MAX_ADDRESSES_FROM_RESOLVER] memory addressesRequired)
    {
        for (uint i = 0; i < resolverAddressesRequired.length; i++) {
            addressesRequired[i] = resolverAddressesRequired[i];
        }
    }

    /* ========== INTERNAL FUNCTIONS ========== */
    function appendToAddressCache(bytes32 name) internal {
        resolverAddressesRequired.push(name);
        require(resolverAddressesRequired.length < MAX_ADDRESSES_FROM_RESOLVER, "Max resolver
cache size met");
        // Because this is designed to be called internally in constructors, we don't
        // check the address exists already in the resolver
        addressCache[name] = resolver.getAddress(name);
    }
}
```

MixinSystemSettings.sol

```
pragma solidity ^0.5.16;

import "./MixinResolver.sol";
```

```
// Internal references
import "./interfaces/IFlexibleStorage.sol";


// https://docs.synthetix.io/contracts/source/contracts/mixinsystemsettings
contract MixinSystemSettings is MixinResolver {
    bytes32 internal constant SETTING_CONTRACT_NAME = "SystemSettings";

    bytes32 internal constant SETTING_WAITING_PERIOD_SECS = "waitingPeriodSecs";
    bytes32 internal constant SETTING_PRICE_DEVIATION_THRESHOLD_FACTOR =
"priceDeviationThresholdFactor";
    bytes32 internal constant SETTING_ISSUANCE_RATIO = "issuanceRatio";
    bytes32 internal constant SETTING_FEE_PERIOD_DURATION = "feePeriodDuration";
    bytes32 internal constant SETTING_TARGET_THRESHOLD = "targetThreshold";
    bytes32 internal constant SETTING_LIQUIDATION_DELAY = "liquidationDelay";
    bytes32 internal constant SETTING_LIQUIDATION_RATIO = "liquidationRatio";
    bytes32 internal constant SETTING_LIQUIDATION_PENALTY = "liquidationPenalty";
    bytes32 internal constant SETTING_RATE_STALE_PERIOD = "rateStalePeriod";
    bytes32 internal constant SETTING_EXCHANGE_FEE_RATE = "exchangeFeeRate";
    bytes32 internal constant SETTING_MINIMUM_STAKE_TIME = "minimumStakeTime";
    bytes32 internal constant SETTING_AGGREGATOR_WARNING_FLAGS = "aggregatorWarningFlags";
    bytes32 internal constant SETTING_TRADING_REWARDS_ENABLED = "tradingRewardsEnabled";
    bytes32 internal constant SETTING_DEBT_SNAPSHOT_STALE_TIME = "debtSnapshotStaleTime";

    bytes32 private constant CONTRACT_FLEXIBLESTORAGE = "FlexibleStorage";

    constructor() internal {
        appendToAddressCache(CONTRACT_FLEXIBLESTORAGE);
    }

    function flexibleStorage() internal view returns (IFlexibleStorage) {
        return IFlexibleStorage(requireAndGetAddress(CONTRACT_FLEXIBLESTORAGE, "Missing
FlexibleStorage address"));
    }

    function getTradingRewardsEnabled() internal view returns (bool) {
        return flexibleStorage().getBoolValue(SETTING_CONTRACT_NAME,
SETTING_TRADING_REWARDS_ENABLED);
    }

    function getWaitingPeriodSecs() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_WAITING_PERIOD_SECS);
    }

    function getPriceDeviationThresholdFactor() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_PRICE_DEVIATION_THRESHOLD_FACTOR);
    }

    function getIssuanceRatio() internal view returns (uint) {
        // lookup on flexible storage directly for gas savings (rather than via SystemSettings)
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME, SETTING_ISSUANCE_RATIO);
    }

    function getFeePeriodDuration() internal view returns (uint) {
        // lookup on flexible storage directly for gas savings (rather than via SystemSettings)
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_FEE_PERIOD_DURATION);
    }

    function getTargetThreshold() internal view returns (uint) {
        // lookup on flexible storage directly for gas savings (rather than via SystemSettings)
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_TARGET_THRESHOLD);
    }

    function getLiquidationDelay() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_LIQUIDATION_DELAY);
    }

    function getLiquidationRatio() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_LIQUIDATION_RATIO);
    }

    function getLiquidationPenalty() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_LIQUIDATION_PENALTY);
    }

    function getRateStalePeriod() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_RATE_STALE_PERIOD);
    }
```

```
    function getExchangeFeeRate(bytes32 currencyKey) internal view returns (uint) {
        return
            flexibleStorage().getUIntValue(
                SETTING_CONTRACT_NAME,
                keccak256(abi.encodePacked(SETTING_EXCHANGE_FEE_RATE, currencyKey))
            );
    }

    function getMinimumStakeTime() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_MINIMUM_STAKE_TIME);
    }

    function getAggregatorWarningFlags() internal view returns (address) {
        return flexibleStorage().getAddressValue(SETTING_CONTRACT_NAME,
SETTING_AGGREGATOR_WARNING_FLAGS);
    }

    function getDebtSnapshotStaleTime() internal view returns (uint) {
        return flexibleStorage().getUIntValue(SETTING_CONTRACT_NAME,
SETTING_DEBT_SNAPSHOT_STALE_TIME);
    }
}
```

MultiCollateralSynth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Synth.sol";


// https://docs.synthetix.io/contracts/source/contracts/multicollateralsynth
contract MultiCollateralSynth is Synth {
    bytes32 public multiCollateralKey;

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        string memory _tokenName,
        string memory _tokenSymbol,
        address _owner,
        bytes32 _currencyKey,
        uint _totalSupply,
        address _resolver,
        bytes32 _multiCollateralKey
    ) public Synth(_proxy, _tokenState, _tokenName, _tokenSymbol, _owner, _currencyKey, _totalSupply,
_resolver) {
        multiCollateralKey = _multiCollateralKey;

        appendToAddressCache(multiCollateralKey);
    }

    /* ========== VIEWS ======================= */

    function multiCollateral() internal view returns (address) {
        return requireAndGetAddress(multiCollateralKey, "Resolver is missing multiCollateral address");
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    /**
     * @notice Function that allows multi Collateral to issue a certain number of synths from an account.
     * @param account Account to issue synths to
     * @param amount Number of synths
     */
    function issue(address account, uint amount) external onlyInternalContracts {
        super._internalIssue(account, amount);
    }

    /**
     * @notice Function that allows multi Collateral to burn a certain number of synths from an account.
     * @param account Account to burn synths from
     * @param amount Number of synths
     */
    function burn(address account, uint amount) external onlyInternalContracts {
        super._internalBurn(account, amount);
    }

    /* ========== MODIFIERS ========== */

    // Contracts directly interacting with multiCollateralSynth to issue and burn
    modifier onlyInternalContracts() {
        bool isFeePool = msg.sender == address(feePool());
```

```
            bool isExchanger = msg.sender == address(exchanger());
            bool isIssuer = msg.sender == address(issuer());
            bool isMultiCollateral = msg.sender == address(multiCollateral());

            require(
                isFeePool || isExchanger || isIssuer || isMultiCollateral,
                "Only FeePool, Exchanger, Issuer or MultiCollateral contracts allowed"
            );
            _;
        }
}
```

Owned.sol

```
pragma solidity ^0.5.16;


// https://docs.synthetix.io/contracts/source/contracts/owned
contract Owned {
    address public owner;
    address public nominatedOwner;

    constructor(address _owner) public {
        require(_owner != address(0), "Owner address cannot be 0");
        owner = _owner;
        emit OwnerChanged(address(0), _owner);
    }

    function nominateNewOwner(address _owner) external onlyOwner {
        nominatedOwner = _owner;
        emit OwnerNominated(_owner);
    }

    function acceptOwnership() external {
        require(msg.sender == nominatedOwner, "You must be nominated before you can accept ownership");
        emit OwnerChanged(owner, nominatedOwner);
        owner = nominatedOwner;
        nominatedOwner = address(0);
    }

    modifier onlyOwner {
        _onlyOwner();
        _;
    }

    function _onlyOwner() private view {
        require(msg.sender == owner, "Only the contract owner may perform this action");
    }

    event OwnerNominated(address newOwner);
    event OwnerChanged(address oldOwner, address newOwner);
}
```

Pausable.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";


// https://docs.synthetix.io/contracts/source/contracts/pausable
contract Pausable is Owned {
    uint public lastPauseTime;
    bool public paused;

    constructor() internal {
        // This contract is abstract, and thus cannot be instantiated directly
        require(owner != address(0), "Owner must be set");
        // Paused will be false, and lastPauseTime will be 0 upon initialisation
    }

    /**
     * @notice Change the paused state of the contract
     * @dev Only the contract owner may call this.
     */
    function setPaused(bool _paused) external onlyOwner {
        // Ensure we're actually changing the state before we do anything
        if (_paused == paused) {
            return;
        }

        // Set our paused state.
        paused = _paused;
```

```
        // If applicable, set the last pause time.
        if (paused) {
            lastPauseTime = now;
        }

        // Let everyone know that our pause state has changed.
        emit PauseChanged(paused);
    }

    event PauseChanged(bool isPaused);

    modifier notPaused {
        require(!paused, "This action cannot be performed while the contract is paused");
        _;
    }
}
```

Proxy.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";

// Internal references
import "./Proxyable.sol";


// https://docs.synthetix.io/contracts/source/contracts/proxy
contract Proxy is Owned {
    Proxyable public target;

    constructor(address _owner) public Owned(_owner) {}

    function setTarget(Proxyable _target) external onlyOwner {
        target = _target;
        emit TargetUpdated(_target);
    }

    function _emit(
        bytes calldata callData,
        uint numTopics,
        bytes32 topic1,
        bytes32 topic2,
        bytes32 topic3,
        bytes32 topic4
    ) external onlyTarget {
        uint size = callData.length;
        bytes memory _callData = callData;

        assembly {
            /* The first 32 bytes of callData contain its length (as specified by the abi).
             * Length is assumed to be a uint256 and therefore maximum of 32 bytes
             * in length. It is also leftpadded to be a multiple of 32 bytes.
             * This means moving call_data across 32 bytes guarantees we correctly access
             * the data itself. */
            switch numTopics
                case 0 {
                    log0(add(_callData, 32), size)
                }
                case 1 {
                    log1(add(_callData, 32), size, topic1)
                }
                case 2 {
                    log2(add(_callData, 32), size, topic1, topic2)
                }
                case 3 {
                    log3(add(_callData, 32), size, topic1, topic2, topic3)
                }
                case 4 {
                    log4(add(_callData, 32), size, topic1, topic2, topic3, topic4)
                }
        }
    }

    // solhint-disable no-complex-fallback
    function() external payable {
        // Mutable call setting Proxyable.messageSender as this is using call not delegatecall
        target.setMessageSender(msg.sender);

        assembly {
            let free_ptr := mload(0x40)
            calldatacopy(free_ptr, 0, calldatasize)

            /* We must explicitly forward ether to the underlying contract as well. */
            let result := call(gas, sload(target_slot), callvalue, free_ptr, calldatasize, 0, 0)
```

```
                returndatacopy(free_ptr, 0, returndatasize)

                if iszero(result) {
                    revert(free_ptr, returndatasize)
                }
                return(free_ptr, returndatasize)
            }
        }

        modifier onlyTarget {
            require(Proxyable(msg.sender) == target, "Must be proxy target");
            _;
        }

        event TargetUpdated(Proxyable newTarget);
}
```

Proxyable.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";

// Internal references
import "./Proxy.sol";


// https://docs.synthetix.io/contracts/source/contracts/proxyable
contract Proxyable is Owned {
    // This contract should be treated like an abstract contract

    /* The proxy this contract exists behind. */
    Proxy public proxy;
    Proxy public integrationProxy;

    /* The caller of the proxy, passed through to this contract.
     * Note that every function using this member must apply the onlyProxy or
     * optionalProxy modifiers, otherwise their invocations can use stale values. */
    address public messageSender;

    constructor(address payable _proxy) internal {
        // This contract is abstract, and thus cannot be instantiated directly
        require(owner != address(0), "Owner must be set");

        proxy = Proxy(_proxy);
        emit ProxyUpdated(_proxy);
    }

    function setProxy(address payable _proxy) external onlyOwner {
        proxy = Proxy(_proxy);
        emit ProxyUpdated(_proxy);
    }

    function setIntegrationProxy(address payable _integrationProxy) external onlyOwner {
        integrationProxy = Proxy(_integrationProxy);
    }

    function setMessageSender(address sender) external onlyProxy {
        messageSender = sender;
    }

    modifier onlyProxy {
        _onlyProxy();
        _;
    }

    function _onlyProxy() private view {
        require(Proxy(msg.sender) == proxy || Proxy(msg.sender) == integrationProxy, "Only the proxy can
call");
    }

    modifier optionalProxy {
        _optionalProxy();
        _;
    }

    function _optionalProxy() private {
        if (Proxy(msg.sender) != proxy && Proxy(msg.sender) != integrationProxy && messageSender !=
msg.sender) {
            messageSender = msg.sender;
        }
    }

    modifier optionalProxy_onlyOwner {
        _optionalProxy_onlyOwner();
```

```
        _;
    }

    // solhint-disable-next-line func-name-mixedcase
    function _optionalProxy_onlyOwner() private {
        if (Proxy(msg.sender) != proxy && Proxy(msg.sender) != integrationProxy && messageSender != msg.sender) {
            messageSender = msg.sender;
        }
        require(messageSender == owner, "Owner only function");
    }

    event ProxyUpdated(address proxyAddress);
}
```

ProxyERC20.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Proxy.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/proxyerc20
contract ProxyERC20 is Proxy, IERC20 {
    constructor(address _owner) public Proxy(_owner) {}

    // ------------- ERC20 Details ------------- //

    function name() public view returns (string memory) {
        // Immutable static call from target contract
        return IERC20(address(target)).name();
    }

    function symbol() public view returns (string memory) {
        // Immutable static call from target contract
        return IERC20(address(target)).symbol();
    }

    function decimals() public view returns (uint8) {
        // Immutable static call from target contract
        return IERC20(address(target)).decimals();
    }

    // ------------- ERC20 Interface ------------- //

    /**
     * @dev Total number of tokens in existence
     */
    function totalSupply() public view returns (uint256) {
        // Immutable static call from target contract
        return IERC20(address(target)).totalSupply();
    }

    /**
     * @dev Gets the balance of the specified address.
     * @param account The address to query the balance of.
     * @return An uint256 representing the amount owned by the passed address.
     */
    function balanceOf(address account) public view returns (uint256) {
        // Immutable static call from target contract
        return IERC20(address(target)).balanceOf(account);
    }

    /**
     * @dev Function to check the amount of tokens that an owner allowed to a spender.
     * @param owner address The address which owns the funds.
     * @param spender address The address which will spend the funds.
     * @return A uint256 specifying the amount of tokens still available for the spender.
     */
    function allowance(address owner, address spender) public view returns (uint256) {
        // Immutable static call from target contract
        return IERC20(address(target)).allowance(owner, spender);
    }

    /**
     * @dev Transfer token for a specified address
     * @param to The address to transfer to.
     * @param value The amount to be transferred.
     */
    function transfer(address to, uint256 value) public returns (bool) {
        // Mutable state call requires the proxy to tell the target who the msg.sender is.
        target.setMessageSender(msg.sender);

        // Forward the ERC20 call to the target contract
```

```
        IERC20(address(target)).transfer(to, value);

        // Event emitting will occur via Synthetix.Proxy._emit()
        return true;
    }

    /**
     * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
     * Beware that changing an allowance with this method brings the risk that someone may use both the old
     * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
     * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     * @param spender The address which will spend the funds.
     * @param value The amount of tokens to be spent.
     */
    function approve(address spender, uint256 value) public returns (bool) {
        // Mutable state call requires the proxy to tell the target who the msg.sender is.
        target.setMessageSender(msg.sender);

        // Forward the ERC20 call to the target contract
        IERC20(address(target)).approve(spender, value);

        // Event emitting will occur via Synthetix.Proxy._emit()
        return true;
    }

    /**
     * @dev Transfer tokens from one address to another
     * @param from address The address which you want to send tokens from
     * @param to address The address which you want to transfer to
     * @param value uint256 the amount of tokens to be transferred
     */
    function transferFrom(
        address from,
        address to,
        uint256 value
    ) public returns (bool) {
        // Mutable state call requires the proxy to tell the target who the msg.sender is.
        target.setMessageSender(msg.sender);

        // Forward the ERC20 call to the target contract
        IERC20(address(target)).transferFrom(from, to, value);

        // Event emitting will occur via Synthetix.Proxy._emit()
        return true;
    }
}
```

PurgeableSynth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Synth.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal References
import "./interfaces/IExchangeRates.sol";


// https://docs.synthetix.io/contracts/source/contracts/purgeablesynth
contract PurgeableSynth is Synth {
    using SafeDecimalMath for uint;

    // The maximum allowed amount of tokenSupply in equivalent zUSD value for this synth to permit purging
    uint public maxSupplyToPurgeInUSD = 100000 * SafeDecimalMath.unit(); // 100,000

    bytes32 private constant CONTRACT_EXRATES = "ExchangeRates";

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        string memory _tokenName,
        string memory _tokenSymbol,
        address payable _owner,
        bytes32 _currencyKey,
        uint _totalSupply,
        address _resolver
    ) public Synth(_proxy, _tokenState, _tokenName, _tokenSymbol, _owner, _currencyKey, _totalSupply,
_resolver) {
        appendToAddressCache(CONTRACT_EXRATES);
    }
```

```
    /* ========== VIEWS ========== */

    function exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(requireAndGetAddress(CONTRACT_EXRATES, "Missing ExchangeRates address"));
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function purge(address[] calldata addresses) external optionalProxy_onlyOwner {
        IExchangeRates exRates = exchangeRates();

        uint maxSupplyToPurge = exRates.effectiveValue("zUSD", maxSupplyToPurgeInUSD, currencyKey);

        // Only allow purge when total supply is lte the max or the rate is frozen in ExchangeRates
        require(
            totalSupply <= maxSupplyToPurge || exRates.rateIsFrozen(currencyKey),
            "Cannot purge as total supply is above threshold and rate is not frozen."
        );

        for (uint i = 0; i < addresses.length; i++) {
            address holder = addresses[i];

            uint amountHeld = tokenState.balanceOf(holder);

            if (amountHeld > 0) {
                exchanger().exchange(holder, currencyKey, amountHeld, "zUSD", holder);
                emitPurged(holder, amountHeld);
            }
        }
    }

    /* ========== EVENTS ========== */
    event Purged(address indexed account, uint value);
    bytes32 private constant PURGED_SIG = keccak256("Purged(address,uint256)");

    function emitPurged(address account, uint value) internal {
        proxy._emit(abi.encode(value), 2, PURGED_SIG, addressToBytes32(account), 0, 0);
    }
}
```

ReadProxy.sol

```
pragma solidity ^0.5.16;

import "./Owned.sol";


// solhint-disable payable-fallback

// https://docs.synthetix.io/contracts/source/contracts/readproxy
contract ReadProxy is Owned {
    address public target;

    constructor(address _owner) public Owned(_owner) {}

    function setTarget(address _target) external onlyOwner {
        target = _target;
        emit TargetUpdated(target);
    }

    function() external {
        // The basics of a proxy read call
        // Note that msg.sender in the underlying will always be the address of this contract.
        assembly {
            calldatacopy(0, 0, calldatasize)

            // Use of staticcall - this will revert if the underlying function mutates state
            let result := staticcall(gas, sload(target_slot), 0, calldatasize, 0, 0)
            returndatacopy(0, 0, returndatasize)

            if iszero(result) {
                revert(0, returndatasize)
            }
            return(0, returndatasize)
        }
    }

    event TargetUpdated(address newTarget);
}
```

RealtimeDebtCache.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./DebtCache.sol";


// https://docs.synthetix.io/contracts/source/contracts/realtimedebtcache
contract RealtimeDebtCache is DebtCache {
    constructor(address _owner, address _resolver) public DebtCache(_owner, _resolver) {}

    // Report the current debt values from all cached debt functions, including public variables

    function debtSnapshotStaleTime() external view returns (uint) {
        return uint(-1);
    }

    function cachedDebt() external view returns (uint) {
        (uint currentDebt, ) = _currentDebt();
        return currentDebt;
    }

    function cachedSynthDebt(bytes32 currencyKey) external view returns (uint) {
        bytes32[] memory keyArray = new bytes32[](1);
        keyArray[0] = currencyKey;
        (uint[] memory debts, ) = _currentSynthDebts(keyArray);
        return debts[0];
    }

    function cacheTimestamp() external view returns (uint) {
        return block.timestamp;
    }

    function cacheStale() external view returns (bool) {
        return false;
    }

    function cacheInvalid() external view returns (bool) {
        (, bool invalid) = _currentDebt();
        return invalid;
    }

    function cachedSynthDebts(bytes32[] calldata currencyKeys) external view returns (uint[] memory debtValues)
{
        (uint[] memory debts, ) = _currentSynthDebts(currencyKeys);
        return debts;
    }

    function cacheInfo()
        external
        view
        returns (
            uint debt,
            uint timestamp,
            bool isInvalid,
            bool isStale
        )
    {
        (uint currentDebt, bool invalid) = _currentDebt();
        return (currentDebt, block.timestamp, invalid, false);
    }

    // Stub out all mutative functions as no-ops;
    // since they do nothing, their access restrictions have been dropped

    function purgeCachedSynthDebt(bytes32 currencyKey) external {}

    function takeDebtSnapshot() external {}

    function updateCachedSynthDebts(bytes32[] calldata currencyKeys) external {}

    function updateCachedSynthDebtWithRate(bytes32 currencyKey, uint currencyRate) external {}

    function updateCachedSynthDebtsWithRates(bytes32[] calldata currencyKeys, uint[] calldata currencyRates)
external {}

    function updateDebtCacheValidity(bool currentlyInvalid) external {}
}
```

RewardEscrow.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./interfaces/IRewardEscrow.sol";
```

```
// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IERC20.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/ISynthetix.sol";


// https://docs.synthetix.io/contracts/source/contracts/rewardescrow
contract RewardEscrow is Owned, IRewardEscrow {
    using SafeMath for uint;

    /* The corresponding Synthetix contract. */
    ISynthetix public synthetix;

    IFeePool public feePool;

    /* Lists of (timestamp, quantity) pairs per account, sorted in ascending time order.
     * These are the times at which each given quantity of HZN vests. */
    mapping(address => uint[2][]) public vestingSchedules;

    /* An account's total escrowed synthetix balance to save recomputing this for fee extraction purposes. */
    mapping(address => uint) public totalEscrowedAccountBalance;

    /* An account's total vested reward synthetix. */
    mapping(address => uint) public totalVestedAccountBalance;

    /* The total remaining escrowed balance, for verifying the actual horizon balance of this contract against. */
    uint public totalEscrowedBalance;

    uint internal constant TIME_INDEX = 0;
    uint internal constant QUANTITY_INDEX = 1;

    /* Limit vesting entries to disallow unbounded iteration over vesting schedules.
     * There are 5 years of the supply schedule */
    uint public constant MAX_VESTING_ENTRIES = 52 * 5;

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address _owner,
        ISynthetix _synthetix,
        IFeePool _feePool
    ) public Owned(_owner) {
        synthetix = _synthetix;
        feePool = _feePool;
    }

    /* ========== SETTERS ========== */

    /**
     * @notice set the synthetix contract address as we need to transfer HZN when the user vests
     */
    function setSynthetix(ISynthetix _synthetix) external onlyOwner {
        synthetix = _synthetix;
        emit SynthetixUpdated(address(_synthetix));
    }

    /**
     * @notice set the FeePool contract as it is the only authority to be able to call
     * appendVestingEntry with the onlyFeePool modifer
     */
    function setFeePool(IFeePool _feePool) external onlyOwner {
        feePool = _feePool;
        emit FeePoolUpdated(address(_feePool));
    }

    /* ========== VIEW FUNCTIONS ========== */

    /**
     * @notice A simple alias to totalEscrowedAccountBalance: provides ERC20 balance integration.
     */
    function balanceOf(address account) public view returns (uint) {
        return totalEscrowedAccountBalance[account];
    }

    function _numVestingEntries(address account) internal view returns (uint) {
        return vestingSchedules[account].length;
    }

    /**
     * @notice The number of vesting dates in an account's schedule.
     */
    function numVestingEntries(address account) external view returns (uint) {
        return vestingSchedules[account].length;
    }
```

```
/**
  * @notice Get a particular schedule entry for an account.
  * @return A pair of uints: (timestamp, synthetix quantity).
  */
function getVestingScheduleEntry(address account, uint index) public view returns (uint[2] memory) {
        return vestingSchedules[account][index];
}

/**
  * @notice Get the time at which a given schedule entry will vest.
  */
function getVestingTime(address account, uint index) public view returns (uint) {
        return getVestingScheduleEntry(account, index)[TIME_INDEX];
}

/**
  * @notice Get the quantity of HZN associated with a given schedule entry.
  */
function getVestingQuantity(address account, uint index) public view returns (uint) {
        return getVestingScheduleEntry(account, index)[QUANTITY_INDEX];
}

/**
  * @notice Obtain the index of the next schedule entry that will vest for a given user.
  */
function getNextVestingIndex(address account) public view returns (uint) {
        uint len = _numVestingEntries(account);
        for (uint i = 0; i < len; i++) {
                if (getVestingTime(account, i) != 0) {
                        return i;
                }
        }
        return len;
}

/**
  * @notice Obtain the next schedule entry that will vest for a given user.
  * @return A pair of uints: (timestamp, synthetix quantity). */
function getNextVestingEntry(address account) public view returns (uint[2] memory) {
        uint index = getNextVestingIndex(account);
        if (index == _numVestingEntries(account)) {
                return [uint(0), 0];
        }
        return getVestingScheduleEntry(account, index);
}

/**
  * @notice Obtain the time at which the next schedule entry will vest for a given user.
  */
function getNextVestingTime(address account) external view returns (uint) {
        return getNextVestingEntry(account)[TIME_INDEX];
}

/**
  * @notice Obtain the quantity which the next schedule entry will vest for a given user.
  */
function getNextVestingQuantity(address account) external view returns (uint) {
        return getNextVestingEntry(account)[QUANTITY_INDEX];
}

/**
  * @notice return the full vesting schedule entries vest for a given user.
  * @dev For DApps to display the vesting schedule for the
  * inflationary supply over 5 years. Solidity cant return variable length arrays
  * so this is returning pairs of data. Vesting Time at [0] and quantity at [1] and so on
  */
function checkAccountSchedule(address account) public view returns (uint[520] memory) {
        uint[520] memory _result;
        uint schedules = _numVestingEntries(account);
        for (uint i = 0; i < schedules; i++) {
                uint[2] memory pair = getVestingScheduleEntry(account, i);
                _result[i * 2] = pair[0];
                _result[i * 2 + 1] = pair[1];
        }
        return _result;
}

/* =========== MUTATIVE FUNCTIONS =========== */

function _appendVestingEntry(address account, uint quantity) internal {
        /* No empty or already-passed vesting entries allowed. */
        require(quantity != 0, "Quantity cannot be zero");

        /* There must be enough balance in the contract to provide for the vesting entry. */
        totalEscrowedBalance = totalEscrowedBalance.add(quantity);
        require(
```

```
            totalEscrowedBalance <= IERC20(address(synthetix)).balanceOf(address(this)),
            "Must be enough balance in the contract to provide for the vesting entry"
        );

        /* Disallow arbitrarily long vesting schedules in light of the gas limit. */
        uint scheduleLength = vestingSchedules[account].length;
        require(scheduleLength <= MAX_VESTING_ENTRIES, "Vesting schedule is too long");

        /* Escrow the tokens for 1 year. */
        uint time = now + 52 weeks;

        if (scheduleLength == 0) {
            totalEscrowedAccountBalance[account] = quantity;
        } else {
            /* Disallow adding new vested HZN earlier than the last one.
             * Since entries are only appended, this means that no vesting date can be repeated. */
            require(
                getVestingTime(account, scheduleLength - 1) < time,
                "Cannot add new vested entries earlier than the last one"
            );
            totalEscrowedAccountBalance[account] = totalEscrowedAccountBalance[account].add(quantity);
        }

        vestingSchedules[account].push([time, quantity]);

        emit VestingEntryCreated(account, now, quantity);
    }

    /**
     * @notice Add a new vesting entry at a given time and quantity to an account's schedule.
     * @dev A call to this should accompany a previous successful call to synthetix.transfer(rewardEscrow,
amount),
     * to ensure that when the funds are withdrawn, there is enough balance.
     * Note; although this function could technically be used to produce unbounded
     * arrays, it's only withinn the 4 year period of the weekly inflation schedule.
     * @param account The account to append a new vesting entry to.
     * @param quantity The quantity of HZN that will be escrowed.
     */
    function appendVestingEntry(address account, uint quantity) external onlyFeePool {
        _appendVestingEntry(account, quantity);
    }

    /**
     * @notice Allow a user to withdraw any HZN in their schedule that have vested.
     */
    function vest() external {
        uint numEntries = _numVestingEntries(msg.sender);
        uint total;
        for (uint i = 0; i < numEntries; i++) {
            uint time = getVestingTime(msg.sender, i);
            /* The list is sorted; when we reach the first future time, bail out. */
            if (time > now) {
                break;
            }
            uint qty = getVestingQuantity(msg.sender, i);
            if (qty > 0) {
                vestingSchedules[msg.sender][i] = [0, 0];
                total = total.add(qty);
            }
        }

        if (total != 0) {
            totalEscrowedBalance = totalEscrowedBalance.sub(total);
            totalEscrowedAccountBalance[msg.sender]                                                    =
totalEscrowedAccountBalance[msg.sender].sub(total);
            totalVestedAccountBalance[msg.sender] = totalVestedAccountBalance[msg.sender].add(total);
            IERC20(address(synthetix)).transfer(msg.sender, total);
            emit Vested(msg.sender, now, total);
        }
    }

    /* ========== MODIFIERS ========== */

    modifier onlyFeePool() {
        bool isFeePool = msg.sender == address(feePool);

        require(isFeePool, "Only the FeePool contracts can perform this action");
        _;
    }

    /* ========== EVENTS ========== */

    event SynthetixUpdated(address newSynthetix);

    event FeePoolUpdated(address newFeePool);

    event Vested(address indexed beneficiary, uint time, uint value);
```

```
        event VestingEntryCreated(address indexed beneficiary, uint time, uint value);
}
```

RewardsDistribution.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./interfaces/IRewardsDistribution.sol";

// Libraires
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IERC20.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/IRewardsDistribution.sol";


// https://docs.synthetix.io/contracts/source/contracts/rewardsdistribution
contract RewardsDistribution is Owned, IRewardsDistribution {
        using SafeMath for uint;
        using SafeDecimalMath for uint;

        /**
          * @notice Authorised address able to call distributeRewards
          */
        address public authority;

        /**
          * @notice Address of the Synthetix ProxyERC20
          */
        address public synthetixProxy;

        /**
          * @notice Address of the RewardEscrow contract
          */
        address public rewardEscrow;

        /**
          * @notice Address of the FeePoolProxy
          */
        address public feePoolProxy;

        /**
          * @notice An array of addresses and amounts to send
          */
        DistributionData[] public distributions;

        /**
          * @dev _authority maybe the underlying synthetix contract.
          * Remember to set the authority on a synthetix upgrade
          */
        constructor(
                address _owner,
                address _authority,
                address _synthetixProxy,
                address _rewardEscrow,
                address _feePoolProxy
        ) public Owned(_owner) {
                authority = _authority;
                synthetixProxy = _synthetixProxy;
                rewardEscrow = _rewardEscrow;
                feePoolProxy = _feePoolProxy;
        }

        // ========== EXTERNAL SETTERS ==========

        function setSynthetixProxy(address _synthetixProxy) external onlyOwner {
                synthetixProxy = _synthetixProxy;
        }

        function setRewardEscrow(address _rewardEscrow) external onlyOwner {
                rewardEscrow = _rewardEscrow;
        }

        function setFeePoolProxy(address _feePoolProxy) external onlyOwner {
                feePoolProxy = _feePoolProxy;
        }

        /**
          * @notice Set the address of the contract authorised to call distributeRewards()
          * @param _authority Address of the authorised calling contract.
          */
```

```
    function setAuthority(address _authority) external onlyOwner {
        authority = _authority;
    }

    // ========== EXTERNAL FUNCTIONS ==========

    /**
     * @notice Adds a Rewards DistributionData struct to the distributions
     * array. Any entries here will be iterated and rewards distributed to
     * each address when tokens are sent to this contract and distributeRewards()
     * is called by the autority.
     * @param destination An address to send rewards tokens too
     * @param amount The amount of rewards tokens to send
     */
    function addRewardDistribution(address destination, uint amount) external onlyOwner returns (bool) {
        require(destination != address(0), "Cant add a zero address");
        require(amount != 0, "Cant add a zero amount");

        DistributionData memory rewardsDistribution = DistributionData(destination, amount);
        distributions.push(rewardsDistribution);

        emit RewardDistributionAdded(distributions.length - 1, destination, amount);
        return true;
    }

    /**
     * @notice Deletes a RewardDistribution from the distributions
     * so it will no longer be included in the call to distributeRewards()
     * @param index The index of the DistributionData to delete
     */
    function removeRewardDistribution(uint index) external onlyOwner {
        require(index <= distributions.length - 1, "index out of bounds");

        // shift distributions indexes across
        for (uint i = index; i < distributions.length - 1; i++) {
            distributions[i] = distributions[i + 1];
        }
        distributions.length--;

        // Since this function must shift all later entries down to fill the
        // gap from the one it removed, it could in principle consume an
        // unbounded amount of gas. However, the number of entries will
        // presumably always be very low.
    }

    /**
     * @notice Edits a RewardDistribution in the distributions array.
     * @param index The index of the DistributionData to edit
     * @param destination The destination address. Send the same address to keep or different address to change
it.
     * @param amount The amount of tokens to edit. Send the same number to keep or change the amount of
tokens to send.
     */
    function editRewardDistribution(
        uint index,
        address destination,
        uint amount
    ) external onlyOwner returns (bool) {
        require(index <= distributions.length - 1, "index out of bounds");

        distributions[index].destination = destination;
        distributions[index].amount = amount;

        return true;
    }

    function distributeRewards(uint amount) external returns (bool) {
        require(amount > 0, "Nothing to distribute");
        require(msg.sender == authority, "Caller is not authorised");
        require(rewardEscrow != address(0), "RewardEscrow is not set");
        require(synthetixProxy != address(0), "SynthetixProxy is not set");
        require(feePoolProxy != address(0), "FeePoolProxy is not set");
        require(
            IERC20(synthetixProxy).balanceOf(address(this)) >= amount,
            "RewardsDistribution contract does not have enough tokens to distribute"
        );

        uint remainder = amount;

        // Iterate the array of distributions sending the configured amounts
        for (uint i = 0; i < distributions.length; i++) {
            if (distributions[i].destination != address(0) || distributions[i].amount != 0) {
                remainder = remainder.sub(distributions[i].amount);

                // Transfer the HZN
                IERC20(synthetixProxy).transfer(distributions[i].destination, distributions[i].amount);
```

```
                    // If the contract implements RewardsDistributionRecipient.sol, inform it how many HZN its received.
                    bytes memory payload = abi.encodeWithSignature("notifyRewardAmount(uint256)", distributions[i].amount);

                    // solhint-disable avoid-low-level-calls
                    (bool success, ) = distributions[i].destination.call(payload);

                    if (!success) {
                        // Note: we're ignoring the return value as it will fail for contracts that do not implement RewardsDistributionRecipient.sol
                    }
                }
            }

            // After all ditributions have been sent, send the remainder to the RewardsEscrow contract
            IERC20(synthetixProxy).transfer(rewardEscrow, remainder);

            // Tell the FeePool how much it has to distribute to the stakers
            IFeePool(feePoolProxy).setRewardsToDistribute(remainder);

            emit RewardsDistributed(amount);
            return true;
        }

        /* ========== VIEWS ========== */

        /**
         * @notice Retrieve the length of the distributions array
         */
        function distributionsLength() external view returns (uint) {
            return distributions.length;
        }

        /* ========== Events ========== */

        event RewardDistributionAdded(uint index, address destination, uint amount);
        event RewardsDistributed(uint amount);
}
```

RewardsDistributionRecipient.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";


// https://docs.synthetix.io/contracts/source/contracts/rewardsdistributionrecipient
contract RewardsDistributionRecipient is Owned {
        address public rewardsDistribution;

        function notifyRewardAmount(uint256 reward) external;

        modifier onlyRewardsDistribution() {
            require(msg.sender == rewardsDistribution, "Caller is not RewardsDistribution contract");
            _;
        }

        function setRewardsDistribution(address _rewardsDistribution) external onlyOwner {
            rewardsDistribution = _rewardsDistribution;
        }
}
```

SafeDecimalMath.sol

```
pragma solidity ^0.5.16;

// Libraries
import "openzeppelin-solidity-2.3.0/contracts/math/SafeMath.sol";


// https://docs.synthetix.io/contracts/source/libraries/safedecimalmath
library SafeDecimalMath {
        using SafeMath for uint;

        /* Number of decimal places in the representations. */
        uint8 public constant decimals = 18;
        uint8 public constant highPrecisionDecimals = 27;

        /* The number representing 1.0. */
        uint public constant UNIT = 10**uint(decimals);

        /* The number representing 1.0 for higher fidelity numbers. */
        uint public constant PRECISE_UNIT = 10**uint(highPrecisionDecimals);
```

```
    uint private constant UNIT_TO_HIGH_PRECISION_CONVERSION_FACTOR =
10**uint(highPrecisionDecimals - decimals);

    /**
     * @return Provides an interface to UNIT.
     */
    function unit() external pure returns (uint) {
        return UNIT;
    }

    /**
     * @return Provides an interface to PRECISE_UNIT.
     */
    function preciseUnit() external pure returns (uint) {
        return PRECISE_UNIT;
    }

    /**
     * @return The result of multiplying x and y, interpreting the operands as fixed-point
     * decimals.
     *
     * @dev A unit factor is divided out after the product of x and y is evaluated,
     * so that product must be less than 2**256. As this is an integer division,
     * the internal division always rounds down. This helps save on gas. Rounding
     * is more expensive on gas.
     */
    function multiplyDecimal(uint x, uint y) internal pure returns (uint) {
        /* Divide by UNIT to remove the extra factor introduced by the product. */
        return x.mul(y) / UNIT;
    }

    /**
     * @return The result of safely multiplying x and y, interpreting the operands
     * as fixed-point decimals of the specified precision unit.
     *
     * @dev The operands should be in the form of a the specified unit factor which will be
     * divided out after the product of x and y is evaluated, so that product must be
     * less than 2**256.
     *
     * Unlike multiplyDecimal, this function rounds the result to the nearest increment.
     * Rounding is useful when you need to retain fidelity for small decimal numbers
     * (eg. small fractions or percentages).
     */
    function _multiplyDecimalRound(
        uint x,
        uint y,
        uint precisionUnit
    ) private pure returns (uint) {
        /* Divide by UNIT to remove the extra factor introduced by the product. */
        uint quotientTimesTen = x.mul(y) / (precisionUnit / 10);

        if (quotientTimesTen % 10 >= 5) {
            quotientTimesTen += 10;
        }

        return quotientTimesTen / 10;
    }

    /**
     * @return The result of safely multiplying x and y, interpreting the operands
     * as fixed-point decimals of a precise unit.
     *
     * @dev The operands should be in the precise unit factor which will be
     * divided out after the product of x and y is evaluated, so that product must be
     * less than 2**256.
     *
     * Unlike multiplyDecimal, this function rounds the result to the nearest increment.
     * Rounding is useful when you need to retain fidelity for small decimal numbers
     * (eg. small fractions or percentages).
     */
    function multiplyDecimalRoundPrecise(uint x, uint y) internal pure returns (uint) {
        return _multiplyDecimalRound(x, y, PRECISE_UNIT);
    }

    /**
     * @return The result of safely multiplying x and y, interpreting the operands
     * as fixed-point decimals of a standard unit.
     *
     * @dev The operands should be in the standard unit factor which will be
     * divided out after the product of x and y is evaluated, so that product must be
     * less than 2**256.
     *
     * Unlike multiplyDecimal, this function rounds the result to the nearest increment.
     * Rounding is useful when you need to retain fidelity for small decimal numbers
     * (eg. small fractions or percentages).
     */
    function multiplyDecimalRound(uint x, uint y) internal pure returns (uint) {
```

```
        return _multiplyDecimalRound(x, y, UNIT);
    }

    /**
     * @return The result of safely dividing x and y. The return value is a high
     * precision decimal.
     *
     * @dev y is divided after the product of x and the standard precision unit
     * is evaluated, so the product of x and UNIT must be less than 2**256. As
     * this is an integer division, the result is always rounded down.
     * This helps save on gas. Rounding is more expensive on gas.
     */
    function divideDecimal(uint x, uint y) internal pure returns (uint) {
        /* Reintroduce the UNIT factor that will be divided out by y. */
        return x.mul(UNIT).div(y);
    }

    /**
     * @return The result of safely dividing x and y. The return value is as a rounded
     * decimal in the precision unit specified in the parameter.
     *
     * @dev y is divided after the product of x and the specified precision unit
     * is evaluated, so the product of x and the specified precision unit must
     * be less than 2**256. The result is rounded to the nearest increment.
     */
    function _divideDecimalRound(
        uint x,
        uint y,
        uint precisionUnit
    ) private pure returns (uint) {
        uint resultTimesTen = x.mul(precisionUnit * 10).div(y);

        if (resultTimesTen % 10 >= 5) {
            resultTimesTen += 10;
        }

        return resultTimesTen / 10;
    }

    /**
     * @return The result of safely dividing x and y. The return value is as a rounded
     * standard precision decimal.
     *
     * @dev y is divided after the product of x and the standard precision unit
     * is evaluated, so the product of x and the standard precision unit must
     * be less than 2**256. The result is rounded to the nearest increment.
     */
    function divideDecimalRound(uint x, uint y) internal pure returns (uint) {
        return _divideDecimalRound(x, y, UNIT);
    }

    /**
     * @return The result of safely dividing x and y. The return value is as a rounded
     * high precision decimal.
     *
     * @dev y is divided after the product of x and the high precision unit
     * is evaluated, so the product of x and the high precision unit must
     * be less than 2**256. The result is rounded to the nearest increment.
     */
    function divideDecimalRoundPrecise(uint x, uint y) internal pure returns (uint) {
        return _divideDecimalRound(x, y, PRECISE_UNIT);
    }

    /**
     * @dev Convert a standard decimal representation to a high precision one.
     */
    function decimalToPreciseDecimal(uint i) internal pure returns (uint) {
        return i.mul(UNIT_TO_HIGH_PRECISION_CONVERSION_FACTOR);
    }

    /**
     * @dev Convert a high precision decimal to a standard decimal representation.
     */
    function preciseDecimalToDecimal(uint i) internal pure returns (uint) {
        uint quotientTimesTen = i / (UNIT_TO_HIGH_PRECISION_CONVERSION_FACTOR / 10);

        if (quotientTimesTen % 10 >= 5) {
            quotientTimesTen += 10;
        }

        return quotientTimesTen / 10;
    }
}
```

StakingRewards.sol

```
pragma solidity ^0.5.16;

import "openzeppelin-solidity-2.3.0/contracts/math/Math.sol";
import "openzeppelin-solidity-2.3.0/contracts/math/SafeMath.sol";
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/SafeERC20.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";

// Inheritance
import "./interfaces/IStakingRewards.sol";
import "./RewardsDistributionRecipient.sol";
import "./Pausable.sol";


// https://docs.synthetix.io/contracts/source/contracts/stakingrewards
contract StakingRewards is IStakingRewards, RewardsDistributionRecipient, ReentrancyGuard, Pausable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    /* ========== STATE VARIABLES ========== */

    IERC20 public rewardsToken;
    IERC20 public stakingToken;
    uint256 public periodFinish = 0;
    uint256 public rewardRate = 0;
    uint256 public rewardsDuration = 7 days;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;

    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address _owner,
        address _rewardsDistribution,
        address _rewardsToken,
        address _stakingToken
    ) public Owned(_owner) {
        rewardsToken = IERC20(_rewardsToken);
        stakingToken = IERC20(_stakingToken);
        rewardsDistribution = _rewardsDistribution;
    }

    /* ========== VIEWS ========== */

    function totalSupply() external view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }

    function lastTimeRewardApplicable() public view returns (uint256) {
        return Math.min(block.timestamp, periodFinish);
    }

    function rewardPerToken() public view returns (uint256) {
        if (_totalSupply == 0) {
            return rewardPerTokenStored;
        }
        return
            rewardPerTokenStored.add(
                lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(_totalSupply)
            );
    }

    function earned(address account) public view returns (uint256) {
        return
            _balances[account].mul(rewardPerToken().sub(userRewardPerTokenPaid[account])).div(1e18).add(rewards[account]);
    }

    function getRewardForDuration() external view returns (uint256) {
        return rewardRate.mul(rewardsDuration);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function stake(uint256 amount) external nonReentrant notPaused updateReward(msg.sender) {
        require(amount > 0, "Cannot stake 0");
```

```
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
        emit Staked(msg.sender, amount);
    }

    function withdraw(uint256 amount) public nonReentrant updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        stakingToken.safeTransfer(msg.sender, amount);
        emit Withdrawn(msg.sender, amount);
    }

    function getReward() public nonReentrant updateReward(msg.sender) {
        uint256 reward = rewards[msg.sender];
        if (reward > 0) {
            rewards[msg.sender] = 0;
            rewardsToken.safeTransfer(msg.sender, reward);
            emit RewardPaid(msg.sender, reward);
        }
    }

    function exit() external {
        withdraw(_balances[msg.sender]);
        getReward();
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function notifyRewardAmount(uint256 reward) external onlyRewardsDistribution updateReward(address(0)) {
        if (block.timestamp >= periodFinish) {
            rewardRate = reward.div(rewardsDuration);
        } else {
            uint256 remaining = periodFinish.sub(block.timestamp);
            uint256 leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(rewardsDuration);
        }

        // Ensure the provided reward amount is not more than the balance in the contract.
        // This keeps the reward rate in the right range, preventing overflows due to
        // very high values of rewardRate in the earned and rewardsPerToken functions;
        // Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
        uint balance = rewardsToken.balanceOf(address(this));
        require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");

        lastUpdateTime = block.timestamp;
        periodFinish = block.timestamp.add(rewardsDuration);
        emit RewardAdded(reward);
    }

    // Added to support recovering LP Rewards from other systems such as BAL to be distributed to holders
    function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
        // If it's SNX we have to query the token symbol to ensure its not a proxy or underlying
        bool isHZN = keccak256(bytes("HZN")) == keccak256(bytes(ERC20Detailed(tokenAddress).symbol()));
        // Cannot recover the staking token or the rewards token
        require(
            tokenAddress != address(stakingToken) && tokenAddress != address(rewardsToken) && !isHZN,
            "Cannot withdraw the staking or rewards tokens"
        );
        IERC20(tokenAddress).safeTransfer(owner, tokenAmount);
        emit Recovered(tokenAddress, tokenAmount);
    }

    function setRewardsDuration(uint256 _rewardsDuration) external onlyOwner {
        require(
            block.timestamp > periodFinish,
            "Previous rewards period must be complete before changing the duration for the new period"
        );
        rewardsDuration = _rewardsDuration;
        emit RewardsDurationUpdated(rewardsDuration);
    }

    /* ========== MODIFIERS ========== */

    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }

    /* ========== EVENTS ========== */
```

```
        event RewardAdded(uint256 reward);
        event Staked(address indexed user, uint256 amount);
        event Withdrawn(address indexed user, uint256 amount);
        event RewardPaid(address indexed user, uint256 reward);
        event RewardsDurationUpdated(uint256 newDuration);
        event Recovered(address token, uint256 amount);
}
```

State.sol

```
pragma solidity ^0.5.16;

import "openzeppelin-solidity-2.3.0/contracts/math/Math.sol";
import "openzeppelin-solidity-2.3.0/contracts/math/SafeMath.sol";
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/SafeERC20.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";

// Inheritance
import "./interfaces/IStakingRewards.sol";
import "./RewardsDistributionRecipient.sol";
import "./Pausable.sol";


// https://docs.synthetix.io/contracts/source/contracts/stakingrewards
contract StakingRewards is IStakingRewards, RewardsDistributionRecipient, ReentrancyGuard, Pausable {
        using SafeMath for uint256;
        using SafeERC20 for IERC20;

        /* ========== STATE VARIABLES ========== */

        IERC20 public rewardsToken;
        IERC20 public stakingToken;
        uint256 public periodFinish = 0;
        uint256 public rewardRate = 0;
        uint256 public rewardsDuration = 7 days;
        uint256 public lastUpdateTime;
        uint256 public rewardPerTokenStored;

        mapping(address => uint256) public userRewardPerTokenPaid;
        mapping(address => uint256) public rewards;

        uint256 private _totalSupply;
        mapping(address => uint256) private _balances;

        /* ========== CONSTRUCTOR ========== */

        constructor(
                address _owner,
                address _rewardsDistribution,
                address _rewardsToken,
                address _stakingToken
        ) public Owned(_owner) {
                rewardsToken = IERC20(_rewardsToken);
                stakingToken = IERC20(_stakingToken);
                rewardsDistribution = _rewardsDistribution;
        }

        /* ========== VIEWS ========== */

        function totalSupply() external view returns (uint256) {
                return _totalSupply;
        }

        function balanceOf(address account) external view returns (uint256) {
                return _balances[account];
        }

        function lastTimeRewardApplicable() public view returns (uint256) {
                return Math.min(block.timestamp, periodFinish);
        }

        function rewardPerToken() public view returns (uint256) {
                if (_totalSupply == 0) {
                        return rewardPerTokenStored;
                }
                return
                        rewardPerTokenStored.add(

    lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(_totalSupply)
                        );
        }

        function earned(address account) public view returns (uint256) {
                return
```

```
_balances[account].mul(rewardPerToken().sub(userRewardPerTokenPaid[account])).div(1e18).add(rewards[account]);
    }

    function getRewardForDuration() external view returns (uint256) {
        return rewardRate.mul(rewardsDuration);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function stake(uint256 amount) external nonReentrant notPaused updateReward(msg.sender) {
        require(amount > 0, "Cannot stake 0");
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        stakingToken.safeTransferFrom(msg.sender, address(this), amount);
        emit Staked(msg.sender, amount);
    }

    function withdraw(uint256 amount) public nonReentrant updateReward(msg.sender) {
        require(amount > 0, "Cannot withdraw 0");
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        stakingToken.safeTransfer(msg.sender, amount);
        emit Withdrawn(msg.sender, amount);
    }

    function getReward() public nonReentrant updateReward(msg.sender) {
        uint256 reward = rewards[msg.sender];
        if (reward > 0) {
            rewards[msg.sender] = 0;
            rewardsToken.safeTransfer(msg.sender, reward);
            emit RewardPaid(msg.sender, reward);
        }
    }

    function exit() external {
        withdraw(_balances[msg.sender]);
        getReward();
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function notifyRewardAmount(uint256 reward) external onlyRewardsDistribution updateReward(address(0)) {
        if (block.timestamp >= periodFinish) {
            rewardRate = reward.div(rewardsDuration);
        } else {
            uint256 remaining = periodFinish.sub(block.timestamp);
            uint256 leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(rewardsDuration);
        }

        // Ensure the provided reward amount is not more than the balance in the contract.
        // This keeps the reward rate in the right range, preventing overflows due to
        // very high values of rewardRate in the earned and rewardsPerToken functions;
        // Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
        uint balance = rewardsToken.balanceOf(address(this));
        require(rewardRate <= balance.div(rewardsDuration), "Provided reward too high");

        lastUpdateTime = block.timestamp;
        periodFinish = block.timestamp.add(rewardsDuration);
        emit RewardAdded(reward);
    }

    // Added to support recovering LP Rewards from other systems such as BAL to be distributed to holders
    function recoverERC20(address tokenAddress, uint256 tokenAmount) external onlyOwner {
        // If it's SNX we have to query the token symbol to ensure its not a proxy or underlying
        bool isSNX = (keccak256(bytes("HZN")) == keccak256(bytes(ERC20Detailed(tokenAddress).symbol())));
        // Cannot recover the staking token or the rewards token
        require(
            tokenAddress != address(stakingToken) && tokenAddress != address(rewardsToken) && !isSNX,
            "Cannot withdraw the staking or rewards tokens"
        );
        IERC20(tokenAddress).safeTransfer(owner, tokenAmount);
        emit Recovered(tokenAddress, tokenAmount);
    }

    function setRewardsDuration(uint256 _rewardsDuration) external onlyOwner {
        require(
            block.timestamp > periodFinish,
            "Previous rewards period must be complete before changing the duration for the new period"
        );
        rewardsDuration = _rewardsDuration;
        emit RewardsDurationUpdated(rewardsDuration);
    }

    /* ========== MODIFIERS ========== */
```

```
    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }

    /* ========== EVENTS ========== */

    event RewardAdded(uint256 reward);
    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event RewardPaid(address indexed user, uint256 reward);
    event RewardsDurationUpdated(uint256 newDuration);
    event Recovered(address token, uint256 amount);
}
```

SupplySchedule.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./interfaces/ISupplySchedule.sol";

// Libraries
import "./SafeDecimalMath.sol";
import "./Math.sol";

// Internal references
import "./Proxy.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/supplyschedule
contract SupplySchedule is Owned, ISupplySchedule {
    using SafeMath for uint;
    using SafeDecimalMath for uint;
    using Math for uint;

    // Time of the last inflation supply mint event
    uint public lastMintEvent;

    // Counter for number of weeks since the start of supply inflation
    uint public weekCounter;

    // The number of SNX rewarded to the caller of Synthetix.mint()
    uint public minterReward = 200 * SafeDecimalMath.unit();

    // The initial weekly inflationary supply is 75m / 52 until the start of the decay rate.
    // 75e6 * SafeDecimalMath.unit() / 52
    uint public constant INITIAL_WEEKLY_SUPPLY = 1442307692307692307692307;

    // Address of the SynthetixProxy for the onlySynthetix modifier
    address payable public synthetixProxy;

    // Max SNX rewards for minter
    uint public constant MAX_MINTER_REWARD = 200 * 1e18;

    // How long each inflation period is before mint can be called
    uint public constant MINT_PERIOD_DURATION = 1 weeks;

    uint public constant INFLATION_START_DATE = 1551830400; // 2019-03-06T00:00:00+00:00
    uint public constant MINT_BUFFER = 1 days;
    uint8 public constant SUPPLY_DECAY_START = 40; // Week 40
    uint8 public constant SUPPLY_DECAY_END = 234; //   Supply Decay ends on Week 234 (inclusive of Week
  234 for a total of 195 weeks of inflation decay)

    // Weekly percentage decay of inflationary supply from the first 40 weeks of the 75% inflation rate
    uint public constant DECAY_RATE = 12500000000000000; // 1.25% weekly

    // Percentage growth of terminal supply per annum
    uint public constant TERMINAL_SUPPLY_RATE_ANNUAL = 25000000000000000; // 2.5% pa

    constructor(
        address _owner,
        uint _lastMintEvent,
        uint _currentWeek
    ) public Owned(_owner) {
        lastMintEvent = _lastMintEvent;
        weekCounter = _currentWeek;
```

```
    }

    // ========== VIEWS ==========

    /**
     * @return The amount of SNX mintable for the inflationary supply
     */
    function mintableSupply() external view returns (uint) {
        uint totalAmount;

        if (!isMintable()) {
            return totalAmount;
        }

        uint remainingWeeksToMint = weeksSinceLastIssuance();

        uint currentWeek = weekCounter;

        // Calculate total mintable supply from exponential decay function
        // The decay function stops after week 234
        while (remainingWeeksToMint > 0) {
            currentWeek++;

            if (currentWeek < SUPPLY_DECAY_START) {
                // If current week is before supply decay we add initial supply to mintableSupply
                totalAmount = totalAmount.add(INITIAL_WEEKLY_SUPPLY);
                remainingWeeksToMint--;
            } else if (currentWeek <= SUPPLY_DECAY_END) {
                // if current week before supply decay ends we add the new supply for the week
                // diff between current week and (supply decay start week - 1)
                uint decayCount = currentWeek.sub(SUPPLY_DECAY_START - 1);

                totalAmount = totalAmount.add(tokenDecaySupplyForWeek(decayCount));
                remainingWeeksToMint--;
            } else {
                // Terminal supply is calculated on the total supply of Synthetix including any new supply
                // We can compound the remaining week's supply at the fixed terminal rate
                uint totalSupply = IERC20(synthetixProxy).totalSupply();
                uint currentTotalSupply = totalSupply.add(totalAmount);

                totalAmount = totalAmount.add(terminalInflationSupply(currentTotalSupply,
remainingWeeksToMint));
                remainingWeeksToMint = 0;
            }
        }

        return totalAmount;
    }

    /**
     * @return A unit amount of decaying inflationary supply from the INITIAL_WEEKLY_SUPPLY
     * @dev New token supply reduces by the decay rate each week calculated as supply =
INITIAL_WEEKLY_SUPPLY * ()
     */
    function tokenDecaySupplyForWeek(uint counter) public pure returns (uint) {
        // Apply exponential decay function to number of weeks since
        // start of inflation smoothing to calculate diminishing supply for the week.
        uint effectiveDecay = (SafeDecimalMath.unit().sub(DECAY_RATE)).powDecimal(counter);
        uint supplyForWeek = INITIAL_WEEKLY_SUPPLY.multiplyDecimal(effectiveDecay);

        return supplyForWeek;
    }

    /**
     * @return A unit amount of terminal inflation supply
     * @dev Weekly compound rate based on number of weeks
     */
    function terminalInflationSupply(uint totalSupply, uint numOfWeeks) public pure returns (uint) {
        // rate = (1 + weekly rate) ^ num of weeks
        uint effectiveCompoundRate =
SafeDecimalMath.unit().add(TERMINAL_SUPPLY_RATE_ANNUAL.div(52)).powDecimal(numOfWeeks);

        // return Supply * (effectiveRate - 1) for extra supply to issue based on number of weeks
        return totalSupply.multiplyDecimal(effectiveCompoundRate.sub(SafeDecimalMath.unit()));
    }

    /**
     * @dev Take timeDiff in seconds (Dividend) and MINT_PERIOD_DURATION as (Divisor)
     * @return Calculate the numberOfWeeks since last mint rounded down to 1 week
     */
    function weeksSinceLastIssuance() public view returns (uint) {
        // Get weeks since lastMintEvent
        // If lastMintEvent not set or 0, then start from inflation start date.
        uint timeDiff = lastMintEvent > 0 ? now.sub(lastMintEvent) : now.sub(INFLATION_START_DATE);
        return timeDiff.div(MINT_PERIOD_DURATION);
    }
```

```
    /**
     * @return boolean whether the MINT_PERIOD_DURATION (7 days)
     * has passed since the lastMintEvent.
     * */
    function isMintable() public view returns (bool) {
        if (now - lastMintEvent > MINT_PERIOD_DURATION) {
            return true;
        }
        return false;
    }

    // ========== MUTATIVE FUNCTIONS ==========

    /**
     * @notice Record the mint event from Synthetix by incrementing the inflation
     * week counter for the number of weeks minted (probabaly always 1)
     * and store the time of the event.
     * @param supplyMinted the amount of SNX the total supply was inflated by.
     * */
    function recordMintEvent(uint supplyMinted) external onlySynthetix returns (bool) {
        uint numberOfWeeksIssued = weeksSinceLastIssuance();

        // add number of weeks minted to weekCounter
        weekCounter = weekCounter.add(numberOfWeeksIssued);

        // Update mint event to latest week issued (start date + number of weeks issued * seconds in week)
        // 1 day time buffer is added so inflation is minted after feePeriod closes
        lastMintEvent =
INFLATION_START_DATE.add(weekCounter.mul(MINT_PERIOD_DURATION)).add(MINT_BUFFER);

        emit SupplyMinted(supplyMinted, numberOfWeeksIssued, lastMintEvent, now);
        return true;
    }

    /**
     * @notice Sets the reward amount of SNX for the caller of the public
     * function Synthetix.mint().
     * This incentivises anyone to mint the inflationary supply and the mintr
     * Reward will be deducted from the inflationary supply and sent to the caller.
     * @param amount the amount of SNX to reward the minter.
     * */
    function setMinterReward(uint amount) external onlyOwner {
        require(amount <= MAX_MINTER_REWARD, "Reward cannot exceed max minter reward");
        minterReward = amount;
        emit MinterRewardUpdated(minterReward);
    }

    // ========== SETTERS ========== */

    /**
     * @notice Set the SynthetixProxy should it ever change.
     * SupplySchedule requires Synthetix address as it has the authority
     * to record mint event.
     * */
    function setSynthetixProxy(ISynthetix _synthetixProxy) external onlyOwner {
        require(address(_synthetixProxy) != address(0), "Address cannot be 0");
        synthetixProxy = address(uint160(address(_synthetixProxy)));
        emit SynthetixProxyUpdated(synthetixProxy);
    }

    // ========== MODIFIERS ==========

    /**
     * @notice Only the Synthetix contract is authorised to call this function
     * */
    modifier onlySynthetix() {
        require(
            msg.sender == address(Proxy(address(synthetixProxy)).target()),
            "Only the synthetix contract can perform this action"
        );
        _;
    }

    /* ========== EVENTS ========== */
    /**
     * @notice Emitted when the inflationary supply is minted
     * */
    event SupplyMinted(uint supplyMinted, uint numberOfWeeksIssued, uint lastMintEvent, uint timestamp);

    /**
     * @notice Emitted when the SNX minter reward amount is updated
     * */
    event MinterRewardUpdated(uint newRewardAmount);

    /**
     * @notice Emitted when setSynthetixProxy is called changing the Synthetix Proxy address
     * */
```

```
    event SynthetixProxyUpdated(address newAddress);
}
```

Synth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./ExternStateToken.sol";
import "./MixinResolver.sol";
import "./interfaces/ISynth.sol";
import "./interfaces/IERC20.sol";

// Internal references
import "./interfaces/ISystemStatus.sol";
import "./interfaces/IFeePool.sol";
import "./interfaces/IExchanger.sol";
import "./interfaces/IIssuer.sol";


// https://docs.synthetix.io/contracts/source/contracts/synth
contract Synth is Owned, IERC20, ExternStateToken, MixinResolver, ISynth {
    /* ========== STATE VARIABLES ========== */

    // Currency key which identifies this Synth to the Synthetix system
    bytes32 public currencyKey;

    uint8 public constant DECIMALS = 18;

    // Where fees are pooled in zUSD
    address public constant FEE_ADDRESS = 0xfeEFEEfeefEeFeefEEFEEfEeFeefEEFeeFEEFEeF;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_SYSTEMSTATUS = "SystemStatus";
    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_FEEPOOL = "FeePool";

    bytes32[24] internal addressesToCache = [CONTRACT_SYSTEMSTATUS, CONTRACT_EXCHANGER,
CONTRACT_ISSUER, CONTRACT_FEEPOOL];

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        string memory _tokenName,
        string memory _tokenSymbol,
        address _owner,
        bytes32 _currencyKey,
        uint _totalSupply,
        address _resolver
    )
        public
        ExternStateToken(_proxy, _tokenState, _tokenName, _tokenSymbol, _totalSupply, DECIMALS, _owner)
        MixinResolver(_resolver, addressesToCache)
    {
        require(_proxy != address(0), "_proxy cannot be 0");
        require(_owner != address(0), "_owner cannot be 0");

        currencyKey = _currencyKey;
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function transfer(address to, uint value) public optionalProxy returns (bool) {
        _ensureCanTransfer(messageSender, value);

        // transfers to FEE_ADDRESS will be exchanged into zUSD and recorded as fee
        if (to == FEE_ADDRESS) {
            return _transferToFeeAddress(to, value);
        }

        // transfers to 0x address will be burned
        if (to == address(0)) {
            return _internalBurn(messageSender, value);
        }

        return super._internalTransfer(messageSender, to, value);
    }

    function transferAndSettle(address to, uint value) public optionalProxy returns (bool) {
        // Exchanger.settle ensures synth is active
        (, , uint numEntriesSettled) = exchanger().settle(messageSender, currencyKey);
```

```
        // Save gas instead of calling transferableSynths
        uint balanceAfter = value;

        if (numEntriesSettled > 0) {
            balanceAfter = tokenState.balanceOf(messageSender);
        }

        // Reduce the value to transfer if balance is insufficient after reclaimed
        value = value > balanceAfter ? balanceAfter : value;

        return super._internalTransfer(messageSender, to, value);
    }

    function transferFrom(
        address from,
        address to,
        uint value
    ) public optionalProxy returns (bool) {
        _ensureCanTransfer(from, value);

        return _internalTransferFrom(from, to, value);
    }

    function transferFromAndSettle(
        address from,
        address to,
        uint value
    ) public optionalProxy returns (bool) {
        // Exchanger.settle() ensures synth is active
        (, , uint numEntriesSettled) = exchanger().settle(from, currencyKey);

        // Save gas instead of calling transferableSynths
        uint balanceAfter = value;

        if (numEntriesSettled > 0) {
            balanceAfter = tokenState.balanceOf(from);
        }

        // Reduce the value to transfer if balance is insufficient after reclaimed
        value = value >= balanceAfter ? balanceAfter : value;

        return _internalTransferFrom(from, to, value);
    }

    /**
     * @notice _transferToFeeAddress function
     * non-zUSD synths are exchanged into zUSD via synthInitiatedExchange
     * notify feePool to record amount as fee paid to feePool */
    function _transferToFeeAddress(address to, uint value) internal returns (bool) {
        uint amountInUSD;

        // zUSD can be transferred to FEE_ADDRESS directly
        if (currencyKey == "zUSD") {
            amountInUSD = value;
            super._internalTransfer(messageSender, to, value);
        } else {
            // else exchange synth into zUSD and send to FEE_ADDRESS
            amountInUSD = exchanger().exchange(messageSender, currencyKey, value, "zUSD",
FEE_ADDRESS);
        }

        // Notify feePool to record zUSD to distribute as fees
        feePool().recordFeePaid(amountInUSD);

        return true;
    }

    function issue(address account, uint amount) external onlyInternalContracts {
        _internalIssue(account, amount);
    }

    function burn(address account, uint amount) external onlyInternalContracts {
        _internalBurn(account, amount);
    }

    function _internalIssue(address account, uint amount) internal {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).add(amount));
        totalSupply = totalSupply.add(amount);
        emitTransfer(address(0), account, amount);
        emitIssued(account, amount);
    }

    function _internalBurn(address account, uint amount) internal returns (bool) {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).sub(amount));
        totalSupply = totalSupply.sub(amount);
        emitTransfer(account, address(0), amount);
```

```
        emitBurned(account, amount);

        return true;
    }

    // Allow owner to set the total supply on import.
    function setTotalSupply(uint amount) external optionalProxy_onlyOwner {
        totalSupply = amount;
    }

    /* ========== VIEWS ========== */
    function systemStatus() internal view returns (ISystemStatus) {
        return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEMSTATUS, "Missing SystemStatus address"));
    }

    function feePool() internal view returns (IFeePool) {
        return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL, "Missing FeePool address"));
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function _ensureCanTransfer(address from, uint value) internal view {
        require(exchanger().maxSecsLeftInWaitingPeriod(from, currencyKey) == 0, "Cannot transfer during waiting period");
        require(transferableSynths(from) >= value, "Insufficient balance after any settlement owing");
        systemStatus().requireSynthActive(currencyKey);
    }

    function transferableSynths(address account) public view returns (uint) {
        (uint reclaimAmount, , ) = exchanger().settlementOwing(account, currencyKey);

        // Note: ignoring rebate amount here because a settle() is required in order to
        // allow the transfer to actually work

        uint balance = tokenState.balanceOf(account);

        if (reclaimAmount > balance) {
            return 0;
        } else {
            return balance.sub(reclaimAmount);
        }
    }

    /* ========== INTERNAL FUNCTIONS ========== */

    function _internalTransferFrom(
        address from,
        address to,
        uint value
    ) internal returns (bool) {
        // Skip allowance update in case of infinite allowance
        if (tokenState.allowance(from, messageSender) != uint(-1)) {
            // Reduce the allowance by the amount we're transferring.
            // The safeSub call will handle an insufficient allowance.
            tokenState.setAllowance(from, messageSender, tokenState.allowance(from, messageSender).sub(value));
        }

        return super._internalTransfer(from, to, value);
    }

    /* ========== MODIFIERS ========== */

    modifier onlyInternalContracts() {
        bool isFeePool = msg.sender == address(feePool());
        bool isExchanger = msg.sender == address(exchanger());
        bool isIssuer = msg.sender == address(issuer());

        require(isFeePool || isExchanger || isIssuer, "Only FeePool, Exchanger or Issuer contracts allowed");
        _;
    }

    /* ========== EVENTS ========== */
    event Issued(address indexed account, uint value);
    bytes32 private constant ISSUED_SIG = keccak256("Issued(address,uint256)");

    function emitIssued(address account, uint value) internal {
        proxy._emit(abi.encode(value), 2, ISSUED_SIG, addressToBytes32(account), 0, 0);
    }
```

```
    event Burned(address indexed account, uint value);
    bytes32 private constant BURNED_SIG = keccak256("Burned(address,uint256)");

    function emitBurned(address account, uint value) internal {
        proxy._emit(abi.encode(value), 2, BURNED_SIG, addressToBytes32(account), 0, 0);
    }
}
```

Synthetix.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./BaseSynthetix.sol";


// https://docs.synthetix.io/contracts/source/contracts/synthetix
contract Synthetix is BaseSynthetix {
    // ========== CONSTRUCTOR ==========

    constructor(
        address payable _proxy,
        TokenState _tokenState,
        address _owner,
        uint _totalSupply,
        address _resolver
    ) public BaseSynthetix(_proxy, _tokenState, _owner, _totalSupply, _resolver) {}

    // ========== OVERRIDDEN FUNCTIONS ==========

    function exchange(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external exchangeActive(sourceCurrencyKey, destinationCurrencyKey) optionalProxy returns (uint
amountReceived) {
        return exchanger().exchange(messageSender, sourceCurrencyKey, sourceAmount,
destinationCurrencyKey, messageSender);
    }

    function exchangeOnBehalf(
        address exchangeForAddress,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey
    ) external exchangeActive(sourceCurrencyKey, destinationCurrencyKey) optionalProxy returns (uint
amountReceived) {
        return
            exchanger().exchangeOnBehalf(
                exchangeForAddress,
                messageSender,
                sourceCurrencyKey,
                sourceAmount,
                destinationCurrencyKey
            );
    }

    function exchangeWithTracking(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external exchangeActive(sourceCurrencyKey, destinationCurrencyKey) optionalProxy returns (uint
amountReceived) {
        return
            exchanger().exchangeWithTracking(
                messageSender,
                sourceCurrencyKey,
                sourceAmount,
                destinationCurrencyKey,
                messageSender,
                originator,
                trackingCode
            );
    }

    function exchangeOnBehalfWithTracking(
        address exchangeForAddress,
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        address originator,
        bytes32 trackingCode
    ) external exchangeActive(sourceCurrencyKey, destinationCurrencyKey) optionalProxy returns (uint
amountReceived) {
```

```
        return
            exchanger().exchangeOnBehalfWithTracking(
                exchangeForAddress,
                messageSender,
                sourceCurrencyKey,
                sourceAmount,
                destinationCurrencyKey,
                originator,
                trackingCode
            );
    }

    function exchangeWithVirtual(
        bytes32 sourceCurrencyKey,
        uint sourceAmount,
        bytes32 destinationCurrencyKey,
        bytes32 trackingCode
    )
        external
        exchangeActive(sourceCurrencyKey, destinationCurrencyKey)
        optionalProxy
        returns (uint amountReceived, IVirtualSynth vSynth)
    {
        return
            exchanger().exchangeWithVirtual(
                messageSender,
                sourceCurrencyKey,
                sourceAmount,
                destinationCurrencyKey,
                messageSender,
                trackingCode
            );
    }

    function settle(bytes32 currencyKey)
        external
        optionalProxy
        returns (
            uint reclaimed,
            uint refunded,
            uint numEntriesSettled
        )
    {
        return exchanger().settle(messageSender, currencyKey);
    }

    function mint() external issuanceActive returns (bool) {
        require(address(rewardsDistribution()) != address(0), "RewardsDistribution not set");

        SupplySchedule _supplySchedule = supplySchedule();
        IRewardsDistribution _rewardsDistribution = rewardsDistribution();

        uint supplyToMint = _supplySchedule.mintableSupply();
        require(supplyToMint > 0, "No supply is mintable");

        // record minting event before mutation to token supply
        _supplySchedule.recordMintEvent(supplyToMint);

        // Set minted HZN balance to RewardEscrow's balance
        // Minus the minterReward and set balance of minter to add reward
        uint minterReward = _supplySchedule.minterReward();
        // Get the remainder
        uint amountToDistribute = supplyToMint.sub(minterReward);

        // Set the token balance to the RewardsDistribution contract
        tokenState.setBalanceOf(
            address(_rewardsDistribution),
            tokenState.balanceOf(address(_rewardsDistribution)).add(amountToDistribute)
        );
        emitTransfer(address(this), address(_rewardsDistribution), amountToDistribute);

        // Kick off the distribution of rewards
        _rewardsDistribution.distributeRewards(amountToDistribute);

        // Assign the minters reward.
        tokenState.setBalanceOf(msg.sender, tokenState.balanceOf(msg.sender).add(minterReward));
        emitTransfer(address(this), msg.sender, minterReward);

        totalSupply = totalSupply.add(supplyToMint);

        return true;
    }

    function liquidateDelinquentAccount(address account, uint zUSDAmount)
        external
        systemActive
        optionalProxy
```

```
        returns (bool)
    {
        (uint totalRedeemed, uint amountLiquidated) = issuer().liquidateDelinquentAccount(
            account,
            zUSDAmount,
            messageSender
        );

        emitAccountLiquidated(account, totalRedeemed, amountLiquidated, messageSender);

        // Transfer HZN redeemed to messageSender
        // Reverts if amount to redeem is more than balanceOf account, ie due to escrowed balance
        return _transferByProxy(account, messageSender, totalRedeemed);
    }

    // ========== EVENTS ==========
    event SynthExchange(
        address indexed account,
        bytes32 fromCurrencyKey,
        uint256 fromAmount,
        bytes32 toCurrencyKey,
        uint256 toAmount,
        address toAddress
    );
    bytes32 internal constant SYNTHEXCHANGE_SIG = keccak256(
        "SynthExchange(address,bytes32,uint256,bytes32,uint256,address)"
    );

    function emitSynthExchange(
        address account,
        bytes32 fromCurrencyKey,
        uint256 fromAmount,
        bytes32 toCurrencyKey,
        uint256 toAmount,
        address toAddress
    ) external onlyExchanger {
        proxy._emit(
            abi.encode(fromCurrencyKey, fromAmount, toCurrencyKey, toAmount, toAddress),
            2,
            SYNTHEXCHANGE_SIG,
            addressToBytes32(account),
            0,
            0
        );
    }

    event ExchangeTracking(bytes32 indexed trackingCode, bytes32 toCurrencyKey, uint256 toAmount);
    bytes32 internal constant EXCHANGE_TRACKING_SIG =
keccak256("ExchangeTracking(bytes32,bytes32,uint256)");

    function emitExchangeTracking(
        bytes32 trackingCode,
        bytes32 toCurrencyKey,
        uint256 toAmount
    ) external onlyExchanger {
        proxy._emit(abi.encode(toCurrencyKey, toAmount), 2, EXCHANGE_TRACKING_SIG, trackingCode, 0,
0);
    }

    event ExchangeReclaim(address indexed account, bytes32 currencyKey, uint amount);
    bytes32 internal constant EXCHANGERECLAIM_SIG =
keccak256("ExchangeReclaim(address,bytes32,uint256)");

    function emitExchangeReclaim(
        address account,
        bytes32 currencyKey,
        uint256 amount
    ) external onlyExchanger {
        proxy._emit(abi.encode(currencyKey, amount), 2, EXCHANGERECLAIM_SIG,
addressToBytes32(account), 0, 0);
    }

    event ExchangeRebate(address indexed account, bytes32 currencyKey, uint amount);
    bytes32 internal constant EXCHANGEREBATE_SIG =
keccak256("ExchangeRebate(address,bytes32,uint256)");

    function emitExchangeRebate(
        address account,
        bytes32 currencyKey,
        uint256 amount
    ) external onlyExchanger {
        proxy._emit(abi.encode(currencyKey, amount), 2, EXCHANGEREBATE_SIG,
addressToBytes32(account), 0, 0);
    }

    event AccountLiquidated(address indexed account, uint hznRedeemed, uint amountLiquidated, address
liquidator);
```

```
    bytes32 internal constant ACCOUNTLIQUIDATED_SIG = keccak256("AccountLiquidated(address,uint256,uint256,address)");

    function emitAccountLiquidated(
        address account,
        uint256 hznRedeemed,
        uint256 amountLiquidated,
        address liquidator
    ) internal {
        proxy._emit(
            abi.encode(hznRedeemed, amountLiquidated, liquidator),
            2,
            ACCOUNTLIQUIDATED_SIG,
            addressToBytes32(account),
            0,
            0
        );
    }
}
```

SynthetixBridgeToBase.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./interfaces/ISynthetixBridgeToBase.sol";

// Internal references
import "./interfaces/ISynthetix.sol";

// solhint-disable indent
import "@eth-optimism/contracts/build/contracts/iOVM/bridge/iOVM_BaseCrossDomainMessenger.sol";


contract SynthetixBridgeToBase is Owned, MixinResolver, ISynthetixBridgeToBase {
    uint32 private constant CROSS_DOMAIN_MESSAGE_GAS_LIMIT = 3e6; //TODO: make this updateable

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */
    bytes32 private constant CONTRACT_EXT_MESSENGER = "ext:Messenger";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_BASE_SYNTHETIXBRIDGETOOPTIMISM = "base:SynthetixBridgeToOptimism";

    bytes32[24] private addressesToCache = [
        CONTRACT_EXT_MESSENGER,
        CONTRACT_SYNTHETIX,
        CONTRACT_BASE_SYNTHETIXBRIDGETOOPTIMISM
    ];

    // ========== CONSTRUCTOR ==========

    constructor(address _owner, address _resolver) public Owned(_owner) MixinResolver(_resolver, addressesToCache) {}

    //
    // ========== INTERNALS ============

    function messenger() internal view returns (iOVM_BaseCrossDomainMessenger) {
        return iOVM_BaseCrossDomainMessenger(requireAndGetAddress(CONTRACT_EXT_MESSENGER, "Missing Messenger address"));
    }

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function synthetixBridgeToOptimism() internal view returns (address) {
        return requireAndGetAddress(CONTRACT_BASE_SYNTHETIXBRIDGETOOPTIMISM, "Missing Bridge address");
    }

    function onlyAllowFromOptimism() internal view {
        // ensure function only callable from the L2 bridge via messenger (aka relayer)
        iOVM_BaseCrossDomainMessenger _messenger = messenger();
        require(msg.sender == address(_messenger), "Only the relayer can call this");
        require(_messenger.xDomainMessageSender() == synthetixBridgeToOptimism(), "Only the L1 bridge can invoke");
    }

    modifier onlyOptimismBridge() {
        onlyAllowFromOptimism();
        _;
    }
```

```
    // ========== PUBLIC FUNCTIONS =========

    // invoked by user on L2
    function initiateWithdrawal(uint amount) external {
        // instruct L2 Synthetix to burn this supply
        synthetix().burnSecondary(msg.sender, amount);

        // create message payload for L1
        bytes memory messageData = abi.encodeWithSignature("completeWithdrawal(address,uint256)", msg.sender, amount);

        // relay the message to Bridge on L1 via L2 Messenger
        messenger().sendMessage(synthetixBridgeToOptimism(), messageData, CROSS_DOMAIN_MESSAGE_GAS_LIMIT);

        emit WithdrawalInitiated(msg.sender, amount);
    }

    // ========= RESTRICTED FUNCTIONS ==============

    // invoked by Messenger on L2
    function mintSecondaryFromDeposit(address account, uint amount) external onlyOptimismBridge {
        // now tell Synthetix to mint these tokens, deposited in L1, into the same account for L2
        synthetix().mintSecondary(account, amount);

        emit MintedSecondary(account, amount);
    }

    // invoked by Messenger on L2
    function mintSecondaryFromDepositForRewards(uint amount) external onlyOptimismBridge {
        // now tell Synthetix to mint these tokens, deposited in L1, into reward escrow on L2
        synthetix().mintSecondaryRewards(amount);

        emit MintedSecondaryRewards(amount);
    }

    // ========== EVENTS ==========
    event MintedSecondary(address indexed account, uint amount);
    event MintedSecondaryRewards(uint amount);
    event WithdrawalInitiated(address indexed account, uint amount);
}
```

SynthetixBridgeToOptimism.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./interfaces/ISynthetixBridgeToOptimism.sol";

// Internal references
import "./interfaces/ISynthetix.sol";
import "./interfaces/IERC20.sol";
import "./interfaces/IIssuer.sol";

// solhint-disable indent
import "@eth-optimism/contracts/build/contracts/iOVM/bridge/iOVM_BaseCrossDomainMessenger.sol";


contract SynthetixBridgeToOptimism is Owned, MixinResolver, ISynthetixBridgeToOptimism {
    uint32 private constant CROSS_DOMAIN_MESSAGE_GAS_LIMIT = 3e6; //TODO: from constant to an updateable value

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */
    bytes32 private constant CONTRACT_EXT_MESSENGER = "ext:Messenger";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 private constant CONTRACT_ISSUER = "Issuer";
    bytes32 private constant CONTRACT_REWARDSDISTRIBUTION = "RewardsDistribution";
    bytes32 private constant CONTRACT_OVM_SYNTHETIXBRIDGETOBASE = "ovm:SynthetixBridgeToBase";

    bytes32[24] private addressesToCache = [
        CONTRACT_EXT_MESSENGER,
        CONTRACT_SYNTHETIX,
        CONTRACT_ISSUER,
        CONTRACT_REWARDSDISTRIBUTION,
        CONTRACT_OVM_SYNTHETIXBRIDGETOBASE
    ];

    bool public activated;

    // ========== CONSTRUCTOR ==========

    constructor(address _owner, address _resolver) public Owned(_owner) MixinResolver(_resolver, addressesToCache) {
        activated = true;
```

```
    }

    //
    // ========== INTERNALS ============

    function messenger() internal view returns (iOVM_BaseCrossDomainMessenger) {
        return iOVM_BaseCrossDomainMessenger(requireAndGetAddress(CONTRACT_EXT_MESSENGER,
"Missing Messenger address"));
    }

    function synthetix() internal view returns (ISynthetix) {
        return ISynthetix(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function synthetixERC20() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function issuer() internal view returns (IIssuer) {
        return IIssuer(requireAndGetAddress(CONTRACT_ISSUER, "Missing Issuer address"));
    }

    function rewardsDistribution() internal view returns (address) {
        return requireAndGetAddress(CONTRACT_REWARDSDISTRIBUTION, "Missing RewardsDistribution
address");
    }

    function synthetixBridgeToBase() internal view returns (address) {
        return requireAndGetAddress(CONTRACT_OVM_SYNTHETIXBRIDGETOBASE, "Missing Bridge
address");
    }

    function isActive() internal view {
        require(activated, "Function deactivated");
    }

    function _rewardDeposit(uint amount) internal {
        // create message payload for L2
        bytes memory messageData =
abi.encodeWithSignature("mintSecondaryFromDepositForRewards(uint256)", amount);

        // relay the message to this contract on L2 via L1 Messenger
        messenger().sendMessage(synthetixBridgeToBase(), messageData,
CROSS_DOMAIN_MESSAGE_GAS_LIMIT);

        emit RewardDeposit(msg.sender, amount);
    }

    // ========== MODIFIERS ============

    modifier requireActive() {
        isActive();
        _;
    }

    // ========== PUBLIC FUNCTIONS =========

    // invoked by user on L1
    function deposit(uint amount) external requireActive {
        require(issuer().debtBalanceOf(msg.sender, "zUSD") == 0, "Cannot deposit with debt");

        // now remove their reward escrow
        // Note: escrowSummary would lose the fidelity of the weekly escrows, so this may not be sufficient
        // uint escrowSummary = rewardEscrow().burnForMigration(msg.sender);

        // move the SNX into this contract
        synthetixERC20().transferFrom(msg.sender, address(this), amount);

        // create message payload for L2
        bytes memory messageData = abi.encodeWithSignature("mintSecondaryFromDeposit(address,uint256)",
msg.sender, amount);

        // relay the message to this contract on L2 via L1 Messenger
        messenger().sendMessage(synthetixBridgeToBase(), messageData,
CROSS_DOMAIN_MESSAGE_GAS_LIMIT);

        emit Deposit(msg.sender, amount);
    }

    // invoked by a generous user on L1
    function rewardDeposit(uint amount) external requireActive {
        // move the SNX into this contract
        synthetixERC20().transferFrom(msg.sender, address(this), amount);
        _rewardDeposit(amount);
    }

    // ========== RESTRICTED FUNCTIONS ==============
```

```
    // invoked by Messenger on L1 after L2 waiting period elapses
    function completeWithdrawal(address account, uint amount) external requireActive {
        // ensure function only callable from L2 Bridge via messenger (aka relayer)
        require(msg.sender == address(messenger()), "Only the relayer can call this");
        require(messenger().xDomainMessageSender() == synthetixBridgeToBase(), "Only the L2 bridge can invoke");

        // transfer amount back to user
        synthetixERC20().transfer(account, amount);

        // no escrow actions - escrow remains on L2
        emit WithdrawalCompleted(account, amount);
    }

    // invoked by the owner for migrating the contract to the new version that will allow for withdrawals
    function migrateBridge(address newBridge) external onlyOwner requireActive {
        require(newBridge != address(0), "Cannot migrate to address 0");
        activated = false;

        IERC20 ERC20Synthetix = synthetixERC20();
        // get the current contract balance and transfer it to the new SynthetixL1ToL2Bridge contract
        uint contractBalance = ERC20Synthetix.balanceOf(address(this));
        ERC20Synthetix.transfer(newBridge, contractBalance);

        emit BridgeMigrated(address(this), newBridge, contractBalance);
    }

    // invoked by RewardsDistribution on L1 (takes SNX)
    function notifyRewardAmount(uint256 amount) external requireActive {
        require(msg.sender == address(rewardsDistribution()), "Caller is not RewardsDistribution contract");

        // to be here means I've been given an amount of SNX to distribute onto L2
        _rewardDeposit(amount);
    }

    // ========== EVENTS ==========

    event BridgeMigrated(address oldBridge, address newBridge, uint amount);
    event Deposit(address indexed account, uint amount);
    event RewardDeposit(address indexed account, uint amount);
    event WithdrawalCompleted(address indexed account, uint amount);
}
```

SynthetixEscrow.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./LimitedSetup.sol";
import "./interfaces/IHasBalance.sol";

// Libraires
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/IERC20.sol";
import "./interfaces/ISynthetix.sol";


// https://docs.synthetix.io/contracts/source/contracts/synthetixescrow
contract SynthetixEscrow is Owned, LimitedSetup(8 weeks), IHasBalance {
    using SafeMath for uint;

    /* The corresponding Synthetix contract. */
    ISynthetix public synthetix;

    /* Lists of (timestamp, quantity) pairs per account, sorted in ascending time order.
     * These are the times at which each given quantity of HZN vests. */
    mapping(address => uint[2][]) public vestingSchedules;

    /* An account's total vested synthetix balance to save recomputing this for fee extraction purposes. */
    mapping(address => uint) public totalVestedAccountBalance;

    /* The total remaining vested balance, for verifying the actual synthetix balance of this contract against. */
    uint public totalVestedBalance;

    uint public constant TIME_INDEX = 0;
    uint public constant QUANTITY_INDEX = 1;

    /* Limit vesting entries to disallow unbounded iteration over vesting schedules. */
    uint public constant MAX_VESTING_ENTRIES = 20;

    /* ========== CONSTRUCTOR ========== */
```

```
constructor(address _owner, ISynthetix _synthetix) public Owned(_owner) {
        synthetix = _synthetix;
}

/* ========== SETTERS ========== */

function setSynthetix(ISynthetix _synthetix) external onlyOwner {
        synthetix = _synthetix;
        emit SynthetixUpdated(address(_synthetix));
}

/* ========== VIEW FUNCTIONS ========== */

/**
  * @notice A simple alias to totalVestedAccountBalance: provides ERC20 balance integration.
  */
function balanceOf(address account) public view returns (uint) {
        return totalVestedAccountBalance[account];
}

/**
  * @notice The number of vesting dates in an account's schedule.
  */
function numVestingEntries(address account) public view returns (uint) {
        return vestingSchedules[account].length;
}

/**
  * @notice Get a particular schedule entry for an account.
  * @return A pair of uints: (timestamp, synthetix quantity).
  */
function getVestingScheduleEntry(address account, uint index) public view returns (uint[2] memory) {
        return vestingSchedules[account][index];
}

/**
  * @notice Get the time at which a given schedule entry will vest.
  */
function getVestingTime(address account, uint index) public view returns (uint) {
        return getVestingScheduleEntry(account, index)[TIME_INDEX];
}

/**
  * @notice Get the quantity of SNX associated with a given schedule entry.
  */
function getVestingQuantity(address account, uint index) public view returns (uint) {
        return getVestingScheduleEntry(account, index)[QUANTITY_INDEX];
}

/**
  * @notice Obtain the index of the next schedule entry that will vest for a given user.
  */
function getNextVestingIndex(address account) public view returns (uint) {
        uint len = numVestingEntries(account);
        for (uint i = 0; i < len; i++) {
                if (getVestingTime(account, i) != 0) {
                        return i;
                }
        }
        return len;
}

/**
  * @notice Obtain the next schedule entry that will vest for a given user.
  * @return A pair of uints: (timestamp, horizon quantity). */
function getNextVestingEntry(address account) public view returns (uint[2] memory) {
        uint index = getNextVestingIndex(account);
        if (index == numVestingEntries(account)) {
                return [uint(0), 0];
        }
        return getVestingScheduleEntry(account, index);
}

/**
  * @notice Obtain the time at which the next schedule entry will vest for a given user.
  */
function getNextVestingTime(address account) external view returns (uint) {
        return getNextVestingEntry(account)[TIME_INDEX];
}

/**
  * @notice Obtain the quantity which the next schedule entry will vest for a given user.
  */
function getNextVestingQuantity(address account) external view returns (uint) {
        return getNextVestingEntry(account)[QUANTITY_INDEX];
}
```

```
/* ========== MUTATIVE FUNCTIONS ========== */

/**
  * @notice Destroy the vesting information associated with an account.
  */
function purgeAccount(address account) external onlyOwner onlyDuringSetup {
        delete vestingSchedules[account];
        totalVestedBalance = totalVestedBalance.sub(totalVestedAccountBalance[account]);
        delete totalVestedAccountBalance[account];
}

/**
  * @notice Add a new vesting entry at a given time and quantity to an account's schedule.
  * @dev A call to this should be accompanied by either enough balance already available
  * in this contract, or a corresponding call to synthetix.endow(), to ensure that when
  * the funds are withdrawn, there is enough balance, as well as correctly calculating
  * the fees.
  * This may only be called by the owner during the contract's setup period.
  * Note; although this function could technically be used to produce unbounded
  * arrays, it's only in the foundation's command to add to these lists.
  * @param account The account to append a new vesting entry to.
  * @param time The absolute unix timestamp after which the vested quantity may be withdrawn.
  * @param quantity The quantity of SNX that will vest.
  */
function appendVestingEntry(
        address account,
        uint time,
        uint quantity
) public onlyOwner onlyDuringSetup {
        /* No empty or already-passed vesting entries allowed. */
        require(now < time, "Time must be in the future");
        require(quantity != 0, "Quantity cannot be zero");

        /* There must be enough balance in the contract to provide for the vesting entry. */
        totalVestedBalance = totalVestedBalance.add(quantity);
        require(
                totalVestedBalance <= IERC20(address(synthetix)).balanceOf(address(this)),
                "Must be enough balance in the contract to provide for the vesting entry"
        );

        /* Disallow arbitrarily long vesting schedules in light of the gas limit. */
        uint scheduleLength = vestingSchedules[account].length;
        require(scheduleLength <= MAX_VESTING_ENTRIES, "Vesting schedule is too long");

        if (scheduleLength == 0) {
                totalVestedAccountBalance[account] = quantity;
        } else {
                /* Disallow adding new vested SNX earlier than the last one.
                 * Since entries are only appended, this means that no vesting date can be repeated. */
                require(
                        getVestingTime(account, numVestingEntries(account) - 1) < time,
                        "Cannot add new vested entries earlier than the last one"
                );
                totalVestedAccountBalance[account] = totalVestedAccountBalance[account].add(quantity);
        }

        vestingSchedules[account].push([time, quantity]);
}

/**
  * @notice Construct a vesting schedule to release a quantities of SNX
  * over a series of intervals.
  * @dev Assumes that the quantities are nonzero
  * and that the sequence of timestamps is strictly increasing.
  * This may only be called by the owner during the contract's setup period.
  */
function addVestingSchedule(
        address account,
        uint[] calldata times,
        uint[] calldata quantities
) external onlyOwner onlyDuringSetup {
        for (uint i = 0; i < times.length; i++) {
                appendVestingEntry(account, times[i], quantities[i]);
        }
}

/**
  * @notice Allow a user to withdraw any SNX in their schedule that have vested.
  */
function vest() external {
        uint numEntries = numVestingEntries(msg.sender);
        uint total;
        for (uint i = 0; i < numEntries; i++) {
                uint time = getVestingTime(msg.sender, i);
                /* The list is sorted; when we reach the first future time, bail out. */
                if (time > now) {
                        break;
```

```
                }
                uint qty = getVestingQuantity(msg.sender, i);
                if (qty > 0) {
                    vestingSchedules[msg.sender][i] = [0, 0];
                    total = total.add(qty);
                }
            }

            if (total != 0) {
                totalVestedBalance = totalVestedBalance.sub(total);
                totalVestedAccountBalance[msg.sender] = totalVestedAccountBalance[msg.sender].sub(total);
                IERC20(address(synthetix)).transfer(msg.sender, total);
                emit Vested(msg.sender, now, total);
            }
        }

        /* ========== EVENTS ========== */

        event SynthetixUpdated(address newSynthetix);

        event Vested(address indexed beneficiary, uint time, uint value);
}
```

SynthetixState.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./State.sol";
import "./LimitedSetup.sol";
import "./interfaces/ISynthetixState.sol";

// Libraries
import "./SafeDecimalMath.sol";


// https://docs.synthetix.io/contracts/source/contracts/synthetixstate
contract SynthetixState is Owned, State, LimitedSetup, ISynthetixState {
        using SafeMath for uint;
        using SafeDecimalMath for uint;

        // A struct for handing values associated with an individual user's debt position
        struct IssuanceData {
            // Percentage of the total debt owned at the time
            // of issuance. This number is modified by the global debt
            // delta array. You can figure out a user's exit price and
            // collateralisation ratio using a combination of their initial
            // debt and the slice of global debt delta which applies to them.
            uint initialDebtOwnership;
            // This lets us know when (in relative terms) the user entered
            // the debt pool so we can calculate their exit price and
            // collateralistion ratio
            uint debtEntryIndex;
        }

        // Issued synth balances for individual fee entitlements and exit price calculations
        mapping(address => IssuanceData) public issuanceData;

        // The total count of people that have outstanding issued synths in any flavour
        uint public totalIssuerCount;

        // Global debt pool tracking
        uint[] public debtLedger;

        constructor(address _owner, address _associatedContract)
            public
            Owned(_owner)
            State(_associatedContract)
            LimitedSetup(1 weeks)
        {}

        /* ========== SETTERS ========== */

        /**
         * @notice Set issuance data for an address
         * @dev Only the associated contract may call this.
         * @param account The address to set the data for.
         * @param initialDebtOwnership The initial debt ownership for this address.
         */
        function setCurrentIssuanceData(address account, uint initialDebtOwnership) external
    onlyAssociatedContract {
            issuanceData[account].initialDebtOwnership = initialDebtOwnership;
            issuanceData[account].debtEntryIndex = debtLedger.length;
        }
```

```
    /**
     * @notice Clear issuance data for an address
     * @dev Only the associated contract may call this.
     * @param account The address to clear the data for.
     */
    function clearIssuanceData(address account) external onlyAssociatedContract {
        delete issuanceData[account];
    }

    /**
     * @notice Increment the total issuer count
     * @dev Only the associated contract may call this.
     */
    function incrementTotalIssuerCount() external onlyAssociatedContract {
        totalIssuerCount = totalIssuerCount.add(1);
    }

    /**
     * @notice Decrement the total issuer count
     * @dev Only the associated contract may call this.
     */
    function decrementTotalIssuerCount() external onlyAssociatedContract {
        totalIssuerCount = totalIssuerCount.sub(1);
    }

    /**
     * @notice Append a value to the debt ledger
     * @dev Only the associated contract may call this.
     * @param value The new value to be added to the debt ledger.
     */
    function appendDebtLedgerValue(uint value) external onlyAssociatedContract {
        debtLedger.push(value);
    }

    // /**
    //  * @notice Import issuer data from the old Synthetix contract before multicurrency
    //  * @dev Only callable by the contract owner, and only for 1 week after deployment.
    //  */
    // function importIssuerData(address[] accounts, uint[] zUSDAmounts) external onlyOwner onlyDuringSetup
{
    //     require(accounts.length == zUSDAmounts.length, "Length mismatch");

    //     for (uint8 i = 0; i < accounts.length; i++) {
    //         _addToDebtRegister(accounts[i], zUSDAmounts[i]);
    //     }
    // }

    // /**
    //  * @notice Import issuer data from the old Synthetix contract before multicurrency
    //  * @dev Only used from importIssuerData above, meant to be disposable
    //  */
    // function _addToDebtRegister(address account, uint amount) internal {
    //     // Note: this function's implementation has been removed from the current Synthetix codebase
    //     // as it could only habe been invoked during setup (see importIssuerData) which has since expired.
    //     // There have been changes to the functions it requires, so to ensure compiles, the below has been
removed.
    //     // For the previous implementation, see Synthetix._addToDebtRegister()
    // }

    /* ========== VIEWS ========== */

    /**
     * @notice Retrieve the length of the debt ledger array
     */
    function debtLedgerLength() external view returns (uint) {
        return debtLedger.length;
    }

    /**
     * @notice Retrieve the most recent entry from the debt ledger
     */
    function lastDebtLedgerEntry() external view returns (uint) {
        return debtLedger[debtLedger.length - 1];
    }

    /**
     * @notice Query whether an account has issued and has an outstanding debt balance
     * @param account The address to query for
     */
    function hasIssued(address account) external view returns (bool) {
        return issuanceData[account].initialDebtOwnership > 0;
    }
}
```

SynthUtil.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./interfaces/ISynth.sol";
import "./interfaces/ISynthetix.sol";
import "./interfaces/IExchangeRates.sol";
import "./interfaces/IAddressResolver.sol";
import "./interfaces/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/synthutil
contract SynthUtil {
    IAddressResolver public addressResolverProxy;

    bytes32 internal constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 internal constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 internal constant zUSD = "zUSD";

    constructor(address resolver) public {
        addressResolverProxy = IAddressResolver(resolver);
    }

    function _synthetix() internal view returns (ISynthetix) {
        return ISynthetix(addressResolverProxy.requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing
Horizon address"));
    }

    function _exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(addressResolverProxy.requireAndGetAddress(CONTRACT_EXRATES, "Missing
ExchangeRates address"));
    }

    function totalSynthsInKey(address account, bytes32 currencyKey) external view returns (uint total) {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            total += exchangeRates.effectiveValue(
                synth.currencyKey(),
                IERC20(address(synth)).balanceOf(account),
                currencyKey
            );
        }
        return total;
    }

    function synthsBalances(address account)
        external
        view
        returns (
            bytes32[] memory,
            uint[] memory,
            uint[] memory
        )
    {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        bytes32[] memory currencyKeys = new bytes32[](numSynths);
        uint[] memory balances = new uint[](numSynths);
        uint[] memory zUSDBalances = new uint[](numSynths);
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            currencyKeys[i] = synth.currencyKey();
            balances[i] = IERC20(address(synth)).balanceOf(account);
            zUSDBalances[i] = exchangeRates.effectiveValue(currencyKeys[i], balances[i], zUSD);
        }
        return (currencyKeys, balances, zUSDBalances);
    }

    function frozenSynths() external view returns (bytes32[] memory) {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        bytes32[] memory frozenSynthsKeys = new bytes32[](numSynths);
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            if (exchangeRates.rateIsFrozen(synth.currencyKey())) {
                frozenSynthsKeys[i] = synth.currencyKey();
            }
        }
        return frozenSynthsKeys;
    }

    function synthsRates() external view returns (bytes32[] memory, uint[] memory) {
        bytes32[] memory currencyKeys = _synthetix().availableCurrencyKeys();
```

```
        return (currencyKeys, _exchangeRates().ratesForCurrencies(currencyKeys));
    }

    function synthsTotalSupplies()
        external
        view
        returns (
            bytes32[] memory,
            uint256[] memory,
            uint256[] memory
        )
    {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();

        uint256 numSynths = synthetix.availableSynthCount();
        bytes32[] memory currencyKeys = new bytes32[](numSynths);
        uint256[] memory balances = new uint256[](numSynths);
        uint256[] memory zUSDBalances = new uint256[](numSynths);
        for (uint256 i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            currencyKeys[i] = synth.currencyKey();
            balances[i] = IERC20(address(synth)).totalSupply();
            zUSDBalances[i] = exchangeRates.effectiveValue(currencyKeys[i], balances[i], zUSD);
        }
        return (currencyKeys, balances, zUSDBalances);
    }
}
```

SystemSettings.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./MixinResolver.sol";
import "./MixinSystemSettings.sol";
import "./interfaces/ISystemSettings.sol";

// Libraries
import "./SafeDecimalMath.sol";


// https://docs.synthetix.io/contracts/source/contracts/systemsettings
contract SystemSettings is Owned, MixinResolver, MixinSystemSettings, ISystemSettings {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    // No more synths may be issued than the value of HZN backing them.
    uint public constant MAX_ISSUANCE_RATIO = 1e18;

    // The fee period must be between 1 day and 60 days.
    uint public constant MIN_FEE_PERIOD_DURATION = 1 days;
    uint public constant MAX_FEE_PERIOD_DURATION = 60 days;

    uint public constant MAX_TARGET_THRESHOLD = 50;

    uint public constant MAX_LIQUIDATION_RATIO = 1e18; // 100% issuance ratio

    uint public constant MAX_LIQUIDATION_PENALTY = 1e18 / 4; // Max 25% liquidation penalty / bonus

    uint public constant RATIO_FROM_TARGET_BUFFER = 2e18; // 200% - mininimum buffer between issuance
  ratio and liquidation ratio

    uint public constant MAX_LIQUIDATION_DELAY = 30 days;
    uint public constant MIN_LIQUIDATION_DELAY = 1 days;

    // Exchange fee may not exceed 10%.
    uint public constant MAX_EXCHANGE_FEE_RATE = 1e18 / 10;

    // Minimum Stake time may not exceed 1 weeks.
    uint public constant MAX_MINIMUM_STAKE_TIME = 1 weeks;

    bytes32[24] private addressesToCache = [bytes32(0)];

    constructor(address _owner, address _resolver)
        public
        Owned(_owner)
        MixinResolver(_resolver, addressesToCache)
        MixinSystemSettings()
    {}

    // ========== VIEWS ==========

    // SIP-37 Fee Reclamation
    // The number of seconds after an exchange is executed that must be waited
```

```
    // before settlement.
    function waitingPeriodSecs() external view returns (uint) {
        return getWaitingPeriodSecs();
    }

    // SIP-65 Decentralized Circuit Breaker
    // The factor amount expressed in decimal format
    // E.g. 3e18 = factor 3, meaning movement up to 3x and above or down to 1/3x and below
    function priceDeviationThresholdFactor() external view returns (uint) {
        return getPriceDeviationThresholdFactor();
    }

    // The raio of collateral
    // Expressed in 18 decimals. So 800% cratio is 100/800 = 0.125 (0.125e18)
    function issuanceRatio() external view returns (uint) {
        return getIssuanceRatio();
    }

    // How long a fee period lasts at a minimum. It is required for
    // anyone to roll over the periods, so they are not guaranteed
    // to roll over at exactly this duration, but the contract enforces
    // that they cannot roll over any quicker than this duration.
    function feePeriodDuration() external view returns (uint) {
        return getFeePeriodDuration();
    }

    // Users are unable to claim fees if their collateralisation ratio drifts out of target threshold
    function targetThreshold() external view returns (uint) {
        return getTargetThreshold();
    }

    // SIP-15 Liquidations
    // liquidation time delay after address flagged (seconds)
    function liquidationDelay() external view returns (uint) {
        return getLiquidationDelay();
    }

    // SIP-15 Liquidations
    // issuance ratio when account can be flagged for liquidation (with 18 decimals), e.g 0.5 issuance ratio
    // when flag means 1/0.5 = 200% cratio
    function liquidationRatio() external view returns (uint) {
        return getLiquidationRatio();
    }

    // SIP-15 Liquidations
    // penalty taken away from target of liquidation (with 18 decimals). E.g. 10% is 0.1e18
    function liquidationPenalty() external view returns (uint) {
        return getLiquidationPenalty();
    }

    // How long will the ExchangeRates contract assume the rate of any asset is correct
    function rateStalePeriod() external view returns (uint) {
        return getRateStalePeriod();
    }

    function exchangeFeeRate(bytes32 currencyKey) external view returns (uint) {
        return getExchangeFeeRate(currencyKey);
    }

    function minimumStakeTime() external view returns (uint) {
        return getMinimumStakeTime();
    }

    function debtSnapshotStaleTime() external view returns (uint) {
        return getDebtSnapshotStaleTime();
    }

    function aggregatorWarningFlags() external view returns (address) {
        return getAggregatorWarningFlags();
    }

    // SIP-63 Trading incentives
    // determines if Exchanger records fee entries in TradingRewards
    function tradingRewardsEnabled() external view returns (bool) {
        return getTradingRewardsEnabled();
    }

    // ========== RESTRICTED ==========

    function setTradingRewardsEnabled(bool _tradingRewardsEnabled) external onlyOwner {
        flexibleStorage().setBoolValue(SETTING_CONTRACT_NAME,
SETTING_TRADING_REWARDS_ENABLED, _tradingRewardsEnabled);
        emit TradingRewardsEnabled(_tradingRewardsEnabled);
    }

    function setWaitingPeriodSecs(uint _waitingPeriodSecs) external onlyOwner {
        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME,  SETTING_WAITING_PERIOD_SECS,
```

```
_waitingPeriodSecs);
        emit WaitingPeriodSecsUpdated(_waitingPeriodSecs);
    }

    function setPriceDeviationThresholdFactor(uint _priceDeviationThresholdFactor) external onlyOwner {
        flexibleStorage().setUIntValue(
            SETTING_CONTRACT_NAME,
            SETTING_PRICE_DEVIATION_THRESHOLD_FACTOR,
            _priceDeviationThresholdFactor
        );
        emit PriceDeviationThresholdUpdated(_priceDeviationThresholdFactor);
    }

    function setIssuanceRatio(uint _issuanceRatio) external onlyOwner {
        require(_issuanceRatio <= MAX_ISSUANCE_RATIO, "New issuance ratio cannot exceed
MAX_ISSUANCE_RATIO");
        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_ISSUANCE_RATIO,
_issuanceRatio);
        emit IssuanceRatioUpdated(_issuanceRatio);
    }

    function setFeePeriodDuration(uint _feePeriodDuration) external onlyOwner {
        require(_feePeriodDuration >= MIN_FEE_PERIOD_DURATION, "value <
MIN_FEE_PERIOD_DURATION");
        require(_feePeriodDuration <= MAX_FEE_PERIOD_DURATION, "value >
MAX_FEE_PERIOD_DURATION");

        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_FEE_PERIOD_DURATION,
_feePeriodDuration);

        emit FeePeriodDurationUpdated(_feePeriodDuration);
    }

    function setTargetThreshold(uint _percent) external onlyOwner {
        require(_percent <= MAX_TARGET_THRESHOLD, "Threshold too high");

        uint _targetThreshold = _percent.mul(SafeDecimalMath.unit()).div(100);

        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_TARGET_THRESHOLD,
_targetThreshold);

        emit TargetThresholdUpdated(_targetThreshold);
    }

    function setLiquidationDelay(uint time) external onlyOwner {
        require(time <= MAX_LIQUIDATION_DELAY, "Must be less than 30 days");
        require(time >= MIN_LIQUIDATION_DELAY, "Must be greater than 1 day");

        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_LIQUIDATION_DELAY,
time);

        emit LiquidationDelayUpdated(time);
    }

    // The collateral / issuance ratio ( debt / collateral ) is higher when there is less collateral backing their debt
    // Upper bound liquidationRatio is 1 + penalty (100% + 10% = 110%) to allow collateral value to cover debt
and liquidation penalty
    function setLiquidationRatio(uint _liquidationRatio) external onlyOwner {
        require(
            liquidationRatio <=
MAX_LIQUIDATION_RATIO.divideDecimal(SafeDecimalMath.unit().add(getLiquidationPenalty())),
            "liquidationRatio > MAX_LIQUIDATION_RATIO / (1 + penalty)"
        );

        // MIN_LIQUIDATION_RATIO is a product of target issuance ratio * RATIO_FROM_TARGET_BUFFER
        // Ensures that liquidation ratio is set so that there is a buffer between the issuance ratio and liquidation
ratio.
        uint MIN_LIQUIDATION_RATIO =
getIssuanceRatio().multiplyDecimal(RATIO_FROM_TARGET_BUFFER);
        require(_liquidationRatio >= MIN_LIQUIDATION_RATIO, "liquidationRatio <
MIN_LIQUIDATION_RATIO");

        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_LIQUIDATION_RATIO,
_liquidationRatio);

        emit LiquidationRatioUpdated(_liquidationRatio);
    }

    function setLiquidationPenalty(uint penalty) external onlyOwner {
        require(penalty <= MAX_LIQUIDATION_PENALTY, "penalty > MAX_LIQUIDATION_PENALTY");

        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_LIQUIDATION_PENALTY,
penalty);

        emit LiquidationPenaltyUpdated(penalty);
    }
```

```
    function setRateStalePeriod(uint period) external onlyOwner {
        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_RATE_STALE_PERIOD,
period);

        emit RateStalePeriodUpdated(period);
    }

    function setExchangeFeeRateForSynths(bytes32[] calldata synthKeys, uint256[] calldata exchangeFeeRates)
        external
        onlyOwner
    {
        require(synthKeys.length == exchangeFeeRates.length, "Array lengths dont match");
        for (uint i = 0; i < synthKeys.length; i++) {
            require(exchangeFeeRates[i] <= MAX_EXCHANGE_FEE_RATE,
"MAX_EXCHANGE_FEE_RATE exceeded");
            flexibleStorage().setUIntValue(
                SETTING_CONTRACT_NAME,
                keccak256(abi.encodePacked(SETTING_EXCHANGE_FEE_RATE, synthKeys[i])),
                exchangeFeeRates[i]
            );
            emit ExchangeFeeUpdated(synthKeys[i], exchangeFeeRates[i]);
        }
    }

    function setMinimumStakeTime(uint _seconds) external onlyOwner {
        require(_seconds <= MAX_MINIMUM_STAKE_TIME, "stake time exceed maximum 1 week");
        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME, SETTING_MINIMUM_STAKE_TIME,
_seconds);
        emit MinimumStakeTimeUpdated(_seconds);
    }

    function setDebtSnapshotStaleTime(uint _seconds) external onlyOwner {
        flexibleStorage().setUIntValue(SETTING_CONTRACT_NAME,
SETTING_DEBT_SNAPSHOT_STALE_TIME, _seconds);
        emit DebtSnapshotStaleTimeUpdated(_seconds);
    }

    function setAggregatorWarningFlags(address _flags) external onlyOwner {
        require(_flags != address(0), "Valid address must be given");
        flexibleStorage().setAddressValue(SETTING_CONTRACT_NAME,
SETTING_AGGREGATOR_WARNING_FLAGS, _flags);
        emit AggregatorWarningFlagsUpdated(_flags);
    }

    // ========== EVENTS ==========
    event TradingRewardsEnabled(bool enabled);
    event WaitingPeriodSecsUpdated(uint waitingPeriodSecs);
    event PriceDeviationThresholdUpdated(uint threshold);
    event IssuanceRatioUpdated(uint newRatio);
    event FeePeriodDurationUpdated(uint newFeePeriodDuration);
    event TargetThresholdUpdated(uint newTargetThreshold);
    event LiquidationDelayUpdated(uint newDelay);
    event LiquidationRatioUpdated(uint newRatio);
    event LiquidationPenaltyUpdated(uint newPenalty);
    event RateStalePeriodUpdated(uint rateStalePeriod);
    event ExchangeFeeUpdated(bytes32 synthKey, uint newExchangeFeeRate);
    event MinimumStakeTimeUpdated(uint minimumStakeTime);
    event DebtSnapshotStaleTimeUpdated(uint debtSnapshotStaleTime);
    event AggregatorWarningFlagsUpdated(address flags);
}
```

SystemStatus.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./interfaces/ISystemStatus.sol";


// https://docs.synthetix.io/contracts/source/contracts/systemstatus
contract SystemStatus is Owned, ISystemStatus {
    mapping(bytes32 => mapping(address => Status)) public accessControl;

    uint248 public constant SUSPENSION_REASON_UPGRADE = 1;

    bytes32 public constant SECTION_SYSTEM = "System";
    bytes32 public constant SECTION_ISSUANCE = "Issuance";
    bytes32 public constant SECTION_EXCHANGE = "Exchange";
    bytes32 public constant SECTION_ZASSET = "Zasset";

    Suspension public systemSuspension;

    Suspension public issuanceSuspension;

    Suspension public exchangeSuspension;
```

```
    mapping(bytes32 => Suspension) public synthSuspension;

    constructor(address _owner) public Owned(_owner) {
        _internalUpdateAccessControl(SECTION_SYSTEM, _owner, true, true);
        _internalUpdateAccessControl(SECTION_ISSUANCE, _owner, true, true);
        _internalUpdateAccessControl(SECTION_EXCHANGE, _owner, true, true);
        _internalUpdateAccessControl(SECTION_ZASSET, _owner, true, true);
    }

    /* ========== VIEWS ========== */
    function requireSystemActive() external view {
        _internalRequireSystemActive();
    }

    function requireIssuanceActive() external view {
        // Issuance requires the system be active
        _internalRequireSystemActive();
        require(!issuanceSuspension.suspended, "Issuance is suspended. Operation prohibited");
    }

    function requireExchangeActive() external view {
        // Issuance requires the system be active
        _internalRequireSystemActive();
        require(!exchangeSuspension.suspended, "Exchange is suspended. Operation prohibited");
    }

    function requireSynthActive(bytes32 currencyKey) external view {
        // Synth exchange and transfer requires the system be active
        _internalRequireSystemActive();
        require(!synthSuspension[currencyKey].suspended, "Zasset is suspended. Operation prohibited");
    }

    function requireSynthsActive(bytes32 sourceCurrencyKey, bytes32 destinationCurrencyKey) external view {
        // Synth exchange and transfer requires the system be active
        _internalRequireSystemActive();

        require(
            !synthSuspension[sourceCurrencyKey].suspended
&& !synthSuspension[destinationCurrencyKey].suspended,
            "One or more zassets are suspended. Operation prohibited"
        );
    }

    function isSystemUpgrading() external view returns (bool) {
        return systemSuspension.suspended && systemSuspension.reason ==
SUSPENSION_REASON_UPGRADE;
    }

    function getSynthSuspensions(bytes32[] calldata synths)
        external
        view
        returns (bool[] memory suspensions, uint256[] memory reasons)
    {
        suspensions = new bool[](synths.length);
        reasons = new uint256[](synths.length);

        for (uint i = 0; i < synths.length; i++) {
            suspensions[i] = synthSuspension[synths[i]].suspended;
            reasons[i] = synthSuspension[synths[i]].reason;
        }
    }

    /* ========== MUTATIVE FUNCTIONS ========== */
    function updateAccessControl(
        bytes32 section,
        address account,
        bool canSuspend,
        bool canResume
    ) external onlyOwner {
        _internalUpdateAccessControl(section, account, canSuspend, canResume);
    }

    function suspendSystem(uint256 reason) external {
        _requireAccessToSuspend(SECTION_SYSTEM);
        systemSuspension.suspended = true;
        systemSuspension.reason = uint248(reason);
        emit SystemSuspended(systemSuspension.reason);
    }

    function resumeSystem() external {
        _requireAccessToResume(SECTION_SYSTEM);
        systemSuspension.suspended = false;
        emit SystemResumed(uint256(systemSuspension.reason));
        systemSuspension.reason = 0;
    }
```

```
function suspendIssuance(uint256 reason) external {
        _requireAccessToSuspend(SECTION_ISSUANCE);
        issuanceSuspension.suspended = true;
        issuanceSuspension.reason = uint248(reason);
        emit IssuanceSuspended(reason);
}

function resumeIssuance() external {
        _requireAccessToResume(SECTION_ISSUANCE);
        issuanceSuspension.suspended = false;
        emit IssuanceResumed(uint256(issuanceSuspension.reason));
        issuanceSuspension.reason = 0;
}

function suspendExchange(uint256 reason) external {
        _requireAccessToSuspend(SECTION_EXCHANGE);
        exchangeSuspension.suspended = true;
        exchangeSuspension.reason = uint248(reason);
        emit ExchangeSuspended(reason);
}

function resumeExchange() external {
        _requireAccessToResume(SECTION_EXCHANGE);
        exchangeSuspension.suspended = false;
        emit ExchangeResumed(uint256(exchangeSuspension.reason));
        exchangeSuspension.reason = 0;
}

function suspendSynth(bytes32 currencyKey, uint256 reason) external {
        _requireAccessToSuspend(SECTION_ZASSET);
        synthSuspension[currencyKey].suspended = true;
        synthSuspension[currencyKey].reason = uint248(reason);
        emit SynthSuspended(currencyKey, reason);
}

function resumeSynth(bytes32 currencyKey) external {
        _requireAccessToResume(SECTION_ZASSET);
        emit SynthResumed(currencyKey, uint256(synthSuspension[currencyKey].reason));
        delete synthSuspension[currencyKey];
}

/* ========== INTERNAL FUNCTIONS ========== */

function _requireAccessToSuspend(bytes32 section) internal view {
        require(accessControl[section][msg.sender].canSuspend, "Restricted to access control list");
}

function _requireAccessToResume(bytes32 section) internal view {
        require(accessControl[section][msg.sender].canResume, "Restricted to access control list");
}

function _internalRequireSystemActive() internal view {
        require(
                !systemSuspension.suspended,
                systemSuspension.reason == SUSPENSION_REASON_UPGRADE
                        ? "Horizon is suspended, upgrade in progress... please stand by"
                        : "Horizon is suspended. Operation prohibited"
        );
}

function _internalUpdateAccessControl(
        bytes32 section,
        address account,
        bool canSuspend,
        bool canResume
) internal {
        require(
                section == SECTION_SYSTEM ||
                        section == SECTION_ISSUANCE ||
                        section == SECTION_EXCHANGE ||
                        section == SECTION_ZASSET,
                "Invalid section supplied"
        );
        accessControl[section][account].canSuspend = canSuspend;
        accessControl[section][account].canResume = canResume;
        emit AccessControlUpdated(section, account, canSuspend, canResume);
}

/* ========== EVENTS ========== */

event SystemSuspended(uint256 reason);
event SystemResumed(uint256 reason);

event IssuanceSuspended(uint256 reason);
event IssuanceResumed(uint256 reason);

event ExchangeSuspended(uint256 reason);
```

```
    event ExchangeResumed(uint256 reason);

    event SynthSuspended(bytes32 currencyKey, uint256 reason);
    event SynthResumed(bytes32 currencyKey, uint256 reason);

    event AccessControlUpdated(bytes32 indexed section, address indexed account, bool canSuspend, bool canResume);
}
```

FakeTradingRewards.sol

```
pragma solidity ^0.5.16;

import "../TradingRewards.sol";

import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20Detailed.sol";

import "../interfaces/IExchanger.sol";


contract FakeTradingRewards is TradingRewards {
    IERC20 public _mockSynthetixToken;

    constructor(
        address owner,
        address periodController,
        address resolver,
        address mockSynthetixToken
    ) public TradingRewards(owner, periodController, resolver) {
        _mockSynthetixToken = IERC20(mockSynthetixToken);
    }

    // Synthetix is mocked with an ERC20 token passed via the constructor.
    function synthetix() internal view returns (IERC20) {
        return IERC20(_mockSynthetixToken);
    }

    // Return msg.sender so that onlyExchanger modifier can be bypassed.
    function exchanger() internal view returns (IExchanger) {
        return IExchanger(msg.sender);
    }
}
```

GenericMock.sol

```
// Source adapted from    https://github.com/EthWorks/Doppelganger/blob/master/contracts/Doppelganger.sol

pragma solidity ^0.5.16;


contract GenericMock {
    mapping(bytes4 => bytes) public mockConfig;

    // solhint-disable payable-fallback, no-complex-fallback
    function() external {
        bytes memory ret = mockConfig[msg.sig];
        assembly {
            return(add(ret, 0x20), mload(ret))
        }
    }

    function mockReturns(bytes4 key, bytes calldata value) external {
        mockConfig[key] = value;
    }
}
```

MockAggregatorV2V3.sol

```
pragma solidity ^0.5.16;


interface AggregatorV2V3Interface {
    function latestRound() external view returns (uint256);

    function decimals() external view returns (uint8);

    function getAnswer(uint256 roundId) external view returns (int256);

    function getTimestamp(uint256 roundId) external view returns (uint256);

    function getRoundData(uint80 _roundId)
        external
        view
        returns (
```

```
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        );

    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        );
}


contract MockAggregatorV2V3 is AggregatorV2V3Interface {
    uint80 public roundId = 0;
    uint8 public keyDecimals = 0;

    struct Entry {
        uint80 roundId;
        int256 answer;
        uint256 startedAt;
        uint256 updatedAt;
        uint80 answeredInRound;
    }

    mapping(uint => Entry) public entries;

    constructor() public {}

    // Mock setup function
    function setLatestAnswer(int256 answer, uint256 timestamp) external {
        roundId++;
        entries[roundId] = Entry({
            roundId: roundId,
            answer: answer,
            startedAt: timestamp,
            updatedAt: timestamp,
            answeredInRound: roundId
        });
    }

    function setLatestAnswerWithRound(
        int256 answer,
        uint256 timestamp,
        uint80 _roundId
    ) external {
        roundId = _roundId;
        entries[roundId] = Entry({
            roundId: roundId,
            answer: answer,
            startedAt: timestamp,
            updatedAt: timestamp,
            answeredInRound: roundId
        });
    }

    function setDecimals(uint8 _decimals) external {
        keyDecimals = _decimals;
    }

    function latestRoundData()
        external
        view
        returns (
            uint80,
            int256,
            uint256,
            uint256,
            uint80
        )
    {
        return getRoundData(uint80(latestRound()));
    }

    function latestRound() public view returns (uint256) {
        return roundId;
    }

    function decimals() external view returns (uint8) {
        return keyDecimals;
```

```
    }

    function getAnswer(uint256 _roundId) external view returns (int256) {
        Entry memory entry = entries[_roundId];
        return entry.answer;
    }

    function getTimestamp(uint256 _roundId) external view returns (uint256) {
        Entry memory entry = entries[_roundId];
        return entry.updatedAt;
    }

    function getRoundData(uint80 _roundId)
        public
        view
        returns (
            uint80,
            int256,
            uint256,
            uint256,
            uint80
        )
    {
        Entry memory entry = entries[_roundId];
        // Emulate a Chainlink aggregator
        require(entry.updatedAt > 0, "No data present");
        return (entry.roundId, entry.answer, entry.startedAt, entry.updatedAt, entry.answeredInRound);
    }
}
```

MockBinaryOptionMarket.sol

```
pragma solidity ^0.5.16;

import "../BinaryOption.sol";

import "../SafeDecimalMath.sol";


contract MockBinaryOptionMarket {
    using SafeDecimalMath for uint;

    uint public deposited;
    uint public senderPrice;
    BinaryOption public binaryOption;

    function setDeposited(uint newDeposited) external {
        deposited = newDeposited;
    }

    function setSenderPrice(uint newPrice) external {
        senderPrice = newPrice;
    }

    function exercisableDeposits() external view returns (uint) {
        return deposited;
    }

    function senderPriceAndExercisableDeposits() external view returns (uint price, uint _deposited) {
        return (senderPrice, deposited);
    }

    function deployOption(address initialBidder, uint initialBid) external {
        binaryOption = new BinaryOption(initialBidder, initialBid);
    }

    function claimOptions() external returns (uint) {
        return binaryOption.claim(msg.sender, senderPrice, deposited);
    }

    function exerciseOptions() external {
        deposited -= binaryOption.balanceOf(msg.sender);
        binaryOption.exercise(msg.sender);
    }

    function bid(address bidder, uint newBid) external {
        binaryOption.bid(bidder, newBid);
        deposited += newBid.divideDecimalRound(senderPrice);
    }

    function refund(address bidder, uint newRefund) external {
        binaryOption.refund(bidder, newRefund);
        deposited -= newRefund.divideDecimalRound(senderPrice);
    }

    function expireOption(address payable beneficiary) external {
```

```
        binaryOption.expire(beneficiary);
    }

    function requireActiveAndUnpaused() external pure {
        return;
    }

    event NewOption(BinaryOption newAddress);
}
```

MockBinaryOptionMarketManager.sol

```
pragma solidity ^0.5.16;

import "../BinaryOptionMarket.sol";
import "../AddressResolver.sol";


contract MockBinaryOptionMarketManager {
    BinaryOptionMarket public market;
    bool public paused = false;

    function createMarket(
        AddressResolver resolver,
        address creator,
        uint[2] calldata creatorLimits,
        bytes32 oracleKey,
        uint strikePrice,
        bool refundsEnabled,
        uint[3] calldata times, // [biddingEnd, maturity, expiry]
        uint[2] calldata bids, // [longBid, shortBid]
        uint[3] calldata fees // [poolFee, creatorFee, refundFee]
    ) external {
        market = new BinaryOptionMarket(
            address(this),
            creator,
            creatorLimits,
            oracleKey,
            strikePrice,
            refundsEnabled,
            times,
            bids,
            fees
        );
        market.setResolverAndSyncCache(resolver);
    }

    function decrementTotalDeposited(uint) external pure {
        return;
    }

    function resolveMarket() external {
        market.resolve();
    }

    function durations()
        external
        pure
        returns (
            uint,
            uint,
            uint
        )
    {
        return (60 * 60 * 24, 0, 0);
    }
}
```

MockContractStorage.sol

```
pragma solidity ^0.5.16;

import "../ContractStorage.sol";


contract MockContractStorage is ContractStorage {
    struct SomeEntry {
        uint value;
        bool flag;
    }

    mapping(bytes32 => mapping(bytes32 => SomeEntry)) public entries;

    constructor(address _resolver) public ContractStorage(_resolver) {}
```

```
    function getEntry(bytes32 contractName, bytes32 record) external view returns (uint value, bool flag) {
        SomeEntry storage entry = entries[hashes[contractName]][record];
        return (entry.value, entry.flag);
    }

    function persistEntry(
        bytes32 contractName,
        bytes32 record,
        uint value,
        bool flag
    ) external onlyContract(contractName) {
        entries[_memoizeHash(contractName)][record].value = value;
        entries[_memoizeHash(contractName)][record].flag = flag;
    }
}
```

MockEtherCollateral.sol

```
pragma solidity ^0.5.16;

import "../SafeDecimalMath.sol";


contract MockEtherCollateral {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    uint public totalIssuedSynths;

    constructor() public {}

    // Mock openLoan function
    function openLoan(uint amount) external {
        // Increment totalIssuedSynths
        totalIssuedSynths = totalIssuedSynths.add(amount);
    }

    function closeLoan(uint amount) external {
        // Increment totalIssuedSynths
        totalIssuedSynths = totalIssuedSynths.sub(amount);
    }
}
```

MockExchanger.sol

```
pragma solidity ^0.5.16;

import "../interfaces/ISynthetix.sol";


contract MockExchanger {
    uint256 private _mockReclaimAmount;
    uint256 private _mockRefundAmount;
    uint256 private _mockNumEntries;
    uint256 private _mockMaxSecsLeft;

    ISynthetix public synthetix;

    constructor(ISynthetix _synthetix) public {
        synthetix = _synthetix;
    }

    // Mock settle function
    function settle(address from, bytes32 currencyKey)
        external
        returns (
            uint256 reclaimed,
            uint256 refunded,
            uint numEntriesSettled
        )
    {
        if (_mockReclaimAmount > 0) {
            synthetix.synths(currencyKey).burn(from, _mockReclaimAmount);
        }

        if (_mockRefundAmount > 0) {
            synthetix.synths(currencyKey).issue(from, _mockRefundAmount);
        }

        _mockMaxSecsLeft = 0;

        return (_mockReclaimAmount, _mockRefundAmount, _mockNumEntries);
    }

    // silence compiler warnings for args
```

```
    function maxSecsLeftInWaitingPeriod(
        address, /* account */
        bytes32 /* currencyKey */
    ) public view returns (uint) {
        return _mockMaxSecsLeft;
    }

    // silence compiler warnings for args
    function settlementOwing(
        address, /* account */
        bytes32 /* currencyKey */
    )
        public
        view
        returns (
            uint,
            uint,
            uint
        )
    {
        return (_mockReclaimAmount, _mockRefundAmount, _mockNumEntries);
    }

    // silence compiler warnings for args
    function hasWaitingPeriodOrSettlementOwing(
        address, /* account */
        bytes32 /* currencyKey */
    ) external view returns (bool) {
        if (_mockMaxSecsLeft > 0) {
            return true;
        }

        if (_mockReclaimAmount > 0 || _mockRefundAmount > 0) {
            return true;
        }

        return false;
    }

    function setReclaim(uint256 _reclaimAmount) external {
        _mockReclaimAmount = _reclaimAmount;
    }

    function setRefund(uint256 _refundAmount) external {
        _mockRefundAmount = _refundAmount;
    }

    function setNumEntries(uint256 _numEntries) external {
        _mockNumEntries = _numEntries;
    }

    function setMaxSecsLeft(uint _maxSecsLeft) external {
        _mockMaxSecsLeft = _maxSecsLeft;
    }
}
```

MockFlagsInterface.sol

```
pragma solidity ^0.5.16;


interface FlagsInterface {
    function getFlag(address) external view returns (bool);

    function getFlags(address[] calldata) external view returns (bool[] memory);
}


contract MockFlagsInterface is FlagsInterface {
    mapping(address => bool) public flags;

    constructor() public {}

    function getFlag(address aggregator) external view returns (bool) {
        return flags[aggregator];
    }

    function getFlags(address[] calldata aggregators) external view returns (bool[] memory results) {
        results = new bool[](aggregators.length);

        for (uint i = 0; i < aggregators.length; i++) {
            results[i] = flags[aggregators[i]];
        }
    }

    function flagAggregator(address aggregator) external {
```

```
        flags[aggregator] = true;
    }

    function unflagAggregator(address aggregator) external {
        flags[aggregator] = false;
    }
}
```

MockMintableSynthetix.sol

```
pragma solidity ^0.5.16;


contract MockMintableSynthetix {
    address public mintSecondaryCallAccount;
    uint public mintSecondaryCallAmount;

    address public burnSecondaryCallAccount;
    uint public burnSecondaryCallAmount;

    function mintSecondary(address account, uint amount) external {
        mintSecondaryCallAccount = account;
        mintSecondaryCallAmount = amount;
    }

    function burnSecondary(address account, uint amount) external {
        burnSecondaryCallAccount = account;
        burnSecondaryCallAmount = amount;
    }
}
```

MockMutator.sol

```
pragma solidity ^0.5.16;


contract MockMutator {
    uint256 public count;

    function read() external view returns (uint) {
        return count;
    }

    function update() external {
        count = count + 1;
    }
}
```

MockRewardsRecipient.sol

```
pragma solidity ^0.5.16;

import "../RewardsDistributionRecipient.sol";
import "../Owned.sol";


contract MockRewardsRecipient is RewardsDistributionRecipient {
    uint256 public rewardsAvailable;

    constructor(address _owner) public Owned(_owner) {}

    function notifyRewardAmount(uint256 reward) external onlyRewardsDistribution {
        rewardsAvailable = rewardsAvailable + reward;
        emit RewardAdded(reward);
    }

    event RewardAdded(uint256 amount);
}
```

MockSynth.sol

```
pragma solidity ^0.5.16;

import "../ExternStateToken.sol";
import "../interfaces/ISystemStatus.sol";


// Mock synth that also adheres to system status

contract MockSynth is ExternStateToken {
    ISystemStatus private systemStatus;
    bytes32 public currencyKey;
```

```
    constructor(
        address payable _proxy,
        TokenState _tokenState,
        string memory _name,
        string memory _symbol,
        uint _totalSupply,
        address _owner,
        bytes32 _currencyKey
    ) public ExternStateToken(_proxy, _tokenState, _name, _symbol, _totalSupply, 18, _owner) {
        currencyKey = _currencyKey;
    }

    // Allow SystemStatus to be passed in directly
    function setSystemStatus(ISystemStatus _status) external {
        systemStatus = _status;
    }

    // Used for PurgeableSynth to test removal
    function setTotalSupply(uint256 _totalSupply) external {
        totalSupply = _totalSupply;
    }

    function transfer(address to, uint value) external optionalProxy returns (bool) {
        systemStatus.requireSynthActive(currencyKey);

        return _transferByProxy(messageSender, to, value);
    }

    function transferFrom(
        address from,
        address to,
        uint value
    ) external optionalProxy returns (bool) {
        systemStatus.requireSynthActive(currencyKey);

        return _transferFromByProxy(messageSender, from, to, value);
    }

    event Issued(address indexed account, uint value);

    event Burned(address indexed account, uint value);

    // Allow these functions which are typically restricted to internal contracts, be open to us for mocking
    function issue(address account, uint amount) external {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).add(amount));
        totalSupply = totalSupply.add(amount);
        emit Issued(account, amount);
    }

    function burn(address account, uint amount) external {
        tokenState.setBalanceOf(account, tokenState.balanceOf(account).sub(amount));
        totalSupply = totalSupply.sub(amount);
        emit Burned(account, amount);
    }
}
```

OneWeekSetup.sol

```
pragma solidity ^0.5.16;

import "../LimitedSetup.sol";


contract OneWeekSetup is LimitedSetup(1 weeks) {
    function testFunc() public view onlyDuringSetup returns (bool) {
        return true;
    }

    function publicSetupExpiryTime() public view returns (uint) {
        return setupExpiryTime;
    }
}
```

PublicEST.sol

```
pragma solidity ^0.5.16;

import "../ExternStateToken.sol";


contract PublicEST is ExternStateToken {
    uint8 public constant DECIMALS = 18;

    constructor(
        address payable _proxy,
```

```
        TokenState _tokenState,
        string memory _name,
        string memory _symbol,
        uint _totalSupply,
        address _owner
    ) public ExternStateToken(_proxy, _tokenState, _name, _symbol, _totalSupply, DECIMALS, _owner) {}

    function transfer(address to, uint value) external optionalProxy returns (bool) {
        return _transferByProxy(messageSender, to, value);
    }

    function transferFrom(
        address from,
        address to,
        uint value
    ) external optionalProxy returns (bool) {
        return _transferFromByProxy(messageSender, from, to, value);
    }

    // Index all parameters to make them easier to find in raw logs (as this will be emitted via a proxy and not decoded)
    event Received(address indexed sender, uint256 indexed inputA, bytes32 indexed inputB);

    function somethingToBeProxied(uint256 inputA, bytes32 inputB) external {
        emit Received(messageSender, inputA, inputB);
    }
}
```

PublicMath.sol

```
/* PublicMath.sol: expose the internal functions in Math library
 * for testing purposes.
 */
pragma solidity ^0.5.16;

import "../Math.sol";


contract PublicMath {
    using Math for uint;

    function powerDecimal(uint x, uint y) public pure returns (uint) {
        return x.powDecimal(y);
    }
}
```

PublicSafeDecimalMath.sol

```
/* PublicSafeDecimalMath.sol: expose the internal functions in SafeDecimalMath
 * for testing purposes.
 */
pragma solidity ^0.5.16;

import "../SafeDecimalMath.sol";


contract PublicSafeDecimalMath {
    using SafeDecimalMath for uint;

    function unit() public pure returns (uint) {
        return SafeDecimalMath.unit();
    }

    function preciseUnit() public pure returns (uint) {
        return SafeDecimalMath.preciseUnit();
    }

    function multiplyDecimal(uint x, uint y) public pure returns (uint) {
        return x.multiplyDecimal(y);
    }

    function multiplyDecimalRound(uint x, uint y) public pure returns (uint) {
        return x.multiplyDecimalRound(y);
    }

    function multiplyDecimalRoundPrecise(uint x, uint y) public pure returns (uint) {
        return x.multiplyDecimalRoundPrecise(y);
    }

    function divideDecimal(uint x, uint y) public pure returns (uint) {
        return x.divideDecimal(y);
    }

    function divideDecimalRound(uint x, uint y) public pure returns (uint) {
        return x.divideDecimalRound(y);
```

```
    }

    function divideDecimalRoundPrecise(uint x, uint y) public pure returns (uint) {
        return x.divideDecimalRoundPrecise(y);
    }

    function decimalToPreciseDecimal(uint i) public pure returns (uint) {
        return i.decimalToPreciseDecimal();
    }

    function preciseDecimalToDecimal(uint i) public pure returns (uint) {
        return i.preciseDecimalToDecimal();
    }
}
```

SwapWithVirtualSynth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20.sol";

// Libraries
import "../SafeDecimalMath.sol";

// Internal references
import "../interfaces/ISynthetix.sol";
import "../interfaces/IAddressResolver.sol";
import "../interfaces/IVirtualSynth.sol";
import "../interfaces/IExchanger.sol";
import {IERC20 as IERC20Detailed} from "../interfaces/IERC20.sol";


interface ICurvePool {
    function exchange(
        int128 i,
        int128 j,
        uint dx,
        uint min_dy
    ) external;
}


contract VirtualToken is ERC20 {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    IVirtualSynth public vSynth;
    ICurvePool public pool;
    IERC20Detailed public targetToken;

    constructor(
        IVirtualSynth _vSynth,
        ICurvePool _pool,
        IERC20Detailed _targetToken
    ) public ERC20() {
        vSynth = _vSynth;
        pool = _pool;
        targetToken = _targetToken;
    }

    function _synthBalance() internal view returns (uint) {
        return IERC20(address(vSynth.synth())).balanceOf(address(this));
    }

    function name() external view returns (string memory) {
        return string(abi.encodePacked("Virtual Token", targetToken.name()));
    }

    function symbol() external view returns (string memory) {
        return string(abi.encodePacked("v", targetToken.symbol()));
    }

    function decimals() external view returns (uint8) {
        return IERC20Detailed(address(vSynth.synth())).decimals();
    }

    function convert(address account, uint amount) external {
        // transfer the vSynth from the creating contract to me
        IERC20(address(vSynth)).transferFrom(msg.sender, address(this), amount);

        // now mint the same supply to the user
        _mint(account, amount);

        emit Converted(address(vSynth), amount);
    }
```

```
    function internalSettle() internal {
        if (vSynth.settled()) {
            return;
        }

        require(vSynth.readyToSettle(), "Not yet ready to settle");

        IERC20 synth = IERC20(address(vSynth.synth()));

        // settle all vSynths for this vToken (now I have synths)
        vSynth.settle(address(this));

        uint balanceAfterSettlement = synth.balanceOf(address(this));

        emit Settled(totalSupply(), balanceAfterSettlement);

        // allow the pool to spend my synths
        synth.approve(address(pool), balanceAfterSettlement);

        // now exchange all my synths (sBTC) for WBTC
        pool.exchange(2, 1, balanceAfterSettlement, 0);
    }

    function settle(address account) external {
        internalSettle();

        uint remainingTokenBalance = targetToken.balanceOf(address(this));

        uint accountBalance = balanceOf(account);

        // now determine how much of the proceeds the user should receive
        uint amount =
  accountBalance.divideDecimalRound(totalSupply()).multiplyDecimalRound(remainingTokenBalance);

        // burn these vTokens
        _burn(account, accountBalance);

        // finally, send the targetToken to the originator
        targetToken.transfer(account, amount);
    }

    event Converted(address indexed virtualSynth, uint amount);
    event Settled(uint totalSupply, uint amountAfterSettled);
}


contract SwapWithVirtualSynth {
    ICurvePool public incomingPool = ICurvePool(0xA5407eAE9Ba41422680e2e00537571bcC53efBfD); //
  Curve: sUSD v2 Swap
    ICurvePool public outgoingPool = ICurvePool(0x7fC77b5c7614E1533320Ea6DDc2Eb61fa00A9714); //
  Curve: sBTC Swap

    ISynthetix public synthetix = ISynthetix(0xC011a73ee8576Fb46F5E1c5751cA3B9Fe0af2a6F);

    IERC20Detailed public sUSD = IERC20Detailed(0x57Ab1ec28D129707052df4dF418D58a2D46d5f51);
    IERC20Detailed public USDC = IERC20Detailed(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    IERC20Detailed public WBTC = IERC20Detailed(0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599);

    function usdcToWBTC(uint amount) external {
        // get user's USDC into this contract
        USDC.transferFrom(msg.sender, address(this), amount);

        // ensure the pool can transferFrom our contract
        USDC.approve(address(incomingPool), amount);

        // now invoke curve USDC to sUSD
        incomingPool.exchange(1, 3, amount, 0);

        // now exchange my sUSD to sBTC
        (, IVirtualSynth vSynth) = synthetix.exchangeWithVirtual("zUSD", sUSD.balanceOf(address(this)),
  "hBTC", bytes32(0));

        // wrap this vSynth in a new token ERC20 contract
        VirtualToken vToken = new VirtualToken(vSynth, outgoingPool, WBTC);

        IERC20 vSynthAsERC20 = IERC20(address(vSynth));

        // get the balance of vSynths I now have
        uint vSynthBalance = vSynthAsERC20.balanceOf(address(this));

        // approve vToken to spend those vSynths
        vSynthAsERC20.approve(address(vToken), vSynthBalance);

        // now have the vToken transfer itself the vSynths and mint the entire vToken supply to the user
        vToken.convert(msg.sender, vSynthBalance);
```

```
            emit VirtualTokenCreated(address(vToken), vSynthBalance);
        }

        event VirtualTokenCreated(address indexed vToken, uint totalSupply);
}
```

TestableAddressSet.sol

```
pragma solidity ^0.5.16;

import "../AddressSetLib.sol";


contract TestableAddressSet {
        using AddressSetLib for AddressSetLib.AddressSet;

        AddressSetLib.AddressSet internal set;

        function contains(address candidate) public view returns (bool) {
                return set.contains(candidate);
        }

        function getPage(uint index, uint pageSize) public view returns (address[] memory) {
                return set.getPage(index, pageSize);
        }

        function add(address element) public {
                set.add(element);
        }

        function remove(address element) public {
                set.remove(element);
        }

        function size() public view returns (uint) {
                return set.elements.length;
        }

        function element(uint index) public view returns (address) {
                return set.elements[index];
        }

        function index(address element) public view returns (uint) {
                return set.indices[element];
        }
}
```

TestableBinaryOptionMarket.sol

```
pragma solidity ^0.5.16;

import "../BinaryOptionMarket.sol";


contract TestableBinaryOptionMarket is BinaryOptionMarket {
        constructor(
                address _owner,
                address _creator,
                uint[2] memory _creatorLimits,
                bytes32 _oracleKey,
                uint256 _strikePrice,
                bool _refundsEnabled,
                uint[3] memory _times,
                uint[2] memory _bids,
                uint[3] memory _fees
        )
                public
                BinaryOptionMarket(_owner, _creator, _creatorLimits, _oracleKey, _strikePrice, _refundsEnabled,
    _times, _bids, _fees)
        {}

        function updatePrices(
                uint256 longBids,
                uint256 shortBids,
                uint totalDebt
        ) public {
                _updatePrices(longBids, shortBids, totalDebt);
        }

        function setManager(address _manager) public {
                owner = _manager;
        }

        function forceClaim(address account) public {
                options.long.claim(account, prices.long, _exercisableDeposits(deposited));
```

```
            options.short.claim(account, prices.short, _exercisableDeposits(deposited));
        }
}
```

TestableDebtCache.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "../DebtCache.sol";


contract TestableDebtCache is DebtCache {
        constructor(address _owner, address _resolver) public DebtCache(_owner, _resolver) {}

        function setCachedSynthDebt(bytes32 currencyKey, uint debt) public {
                _cachedSynthDebt[currencyKey] = debt;
        }
}
```

TestableMixinResolver.sol

```
pragma solidity ^0.5.16;

import "../Owned.sol";
import "../MixinResolver.sol";


contract TestableMixinResolver is Owned, MixinResolver {
        bytes32 private constant CONTRACT_EXAMPLE_1 = "Example_1";
        bytes32 private constant CONTRACT_EXAMPLE_2 = "Example_2";
        bytes32 private constant CONTRACT_EXAMPLE_3 = "Example_3";

        bytes32[24] private addressesToCache = [CONTRACT_EXAMPLE_1, CONTRACT_EXAMPLE_2,
    CONTRACT_EXAMPLE_3];

        constructor(address _owner, address _resolver) public Owned(_owner) MixinResolver(_resolver,
    addressesToCache) {}
}
```

TestablePausable.sol

```
pragma solidity ^0.5.16;

import "../Owned.sol";
import "../Pausable.sol";


/**
  * @title An implementation of Pausable. Used to test the features of the Pausable contract that can only be tested
    by an implementation.
  */
contract TestablePausable is Owned, Pausable {
        uint public someValue;

        constructor(address _owner) public Owned(_owner) Pausable() {}

        function setSomeValue(uint _value) external notPaused {
                someValue = _value;
        }
}
```

TestableState.sol

```
pragma solidity ^0.5.16;

import "../Owned.sol";
import "../State.sol";


contract TestableState is Owned, State {
        constructor(address _owner, address _associatedContract) public Owned(_owner) State(_associatedContract)
    {}

        function testModifier() external onlyAssociatedContract {}
}
```

TokenExchanger.sol

```
/* TokenExchanger.sol: Used for testing contract to contract calls on chain
  * with Synthetix for testing ERC20 compatability
  */
```

```
pragma solidity ^0.5.16;

import "../Owned.sol";
import "../interfaces/ISynthetix.sol";
import "../interfaces/IFeePool.sol";
import "../interfaces/IERC20.sol";


contract TokenExchanger is Owned {
    address public integrationProxy;
    address public synthetix;

    constructor(address _owner, address _integrationProxy) public Owned(_owner) {
        integrationProxy = _integrationProxy;
    }

    function setSynthetixProxy(address _integrationProxy) external onlyOwner {
        integrationProxy = _integrationProxy;
    }

    function setSynthetix(address _synthetix) external onlyOwner {
        synthetix = _synthetix;
    }

    function checkBalance(address account) public view synthetixProxyIsSet returns (uint) {
        return IERC20(integrationProxy).balanceOf(account);
    }

    function checkAllowance(address tokenOwner, address spender) public view synthetixProxyIsSet returns (uint)
{
        return IERC20(integrationProxy).allowance(tokenOwner, spender);
    }

    function checkBalanceSNXDirect(address account) public view synthetixProxyIsSet returns (uint) {
        return IERC20(synthetix).balanceOf(account);
    }

    function getDecimals(address tokenAddress) public view returns (uint) {
        return IERC20(tokenAddress).decimals();
    }

    function doTokenSpend(
        address fromAccount,
        address toAccount,
        uint amount
    ) public synthetixProxyIsSet returns (bool) {
        // Call Immutable static call #1
        require(checkBalance(fromAccount) >= amount, "fromAccount does not have the required balance to
spend");

        // Call Immutable static call #2
        require(
            checkAllowance(fromAccount, address(this)) >= amount,
            "I TokenExchanger, do not have approval to spend this guys tokens"
        );

        // Call Mutable call
        return IERC20(integrationProxy).transferFrom(fromAccount, toAccount, amount);
    }

    modifier synthetixProxyIsSet {
        require(integrationProxy != address(0), "Horizon Integration proxy address not set");
        _;
    }

    event LogString(string name, string value);
    event LogInt(string name, uint value);
    event LogAddress(string name, address value);
    event LogBytes(string name, bytes4 value);
}
```

UsingReadProxy.sol

```
pragma solidity ^0.5.16;

import "../interfaces/IAddressResolver.sol";
import "../interfaces/IExchangeRates.sol";


contract UsingReadProxy {
    IAddressResolver public resolver;

    constructor(IAddressResolver _resolver) public {
        resolver = _resolver;
    }
```

```
        function run(bytes32 currencyKey) external view returns (uint) {
            IExchangeRates exRates = IExchangeRates(resolver.getAddress("ExchangeRates"));
            require(address(exRates) != address(0), "Missing ExchangeRates");
            return exRates.rateForCurrency(currencyKey);
        }
}
```

TokenState.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "./Owned.sol";
import "./State.sol";


// https://docs.synthetix.io/contracts/source/contracts/tokenstate
contract TokenState is Owned, State {
        /* ERC20 fields. */
        mapping(address => uint) public balanceOf;
        mapping(address => mapping(address => uint)) public allowance;

        constructor(address _owner, address _associatedContract) public Owned(_owner) State(_associatedContract)
    {}

        /* ========== SETTERS ========== */

        /**
          * @notice Set ERC20 allowance.
          * @dev Only the associated contract may call this.
          * @param tokenOwner The authorising party.
          * @param spender The authorised party.
          * @param value The total value the authorised party may spend on the
          * authorising party's behalf.
          */
        function setAllowance(
            address tokenOwner,
            address spender,
            uint value
        ) external onlyAssociatedContract {
            allowance[tokenOwner][spender] = value;
        }

        /**
          * @notice Set the balance in a given account
          * @dev Only the associated contract may call this.
          * @param account The account whose value to set.
          * @param value The new balance of the given account.
          */
        function setBalanceOf(address account, uint value) external onlyAssociatedContract {
            balanceOf[account] = value;
        }
}
```

TradingRewards.sol

```
pragma solidity ^0.5.16;

// Internal dependencies.
import "./Pausable.sol";
import "./MixinResolver.sol";
import "./Owned.sol";

// External dependencies.
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/SafeERC20.sol";
import "openzeppelin-solidity-2.3.0/contracts/utils/ReentrancyGuard.sol";

// Libraries.
import "./SafeDecimalMath.sol";

// Internal references.
import "./interfaces/ITradingRewards.sol";
import "./interfaces/IExchanger.sol";


// https://docs.synthetix.io/contracts/source/contracts/tradingrewards
contract TradingRewards is ITradingRewards, ReentrancyGuard, Owned, Pausable, MixinResolver {
        using SafeMath for uint;
        using SafeDecimalMath for uint;
        using SafeERC20 for IERC20;

        /* ========== STATE VARIABLES ========== */

        uint private _currentPeriodID;
```

```
    uint private _balanceAssignedToRewards;
    mapping(uint => Period) private _periods;

    struct Period {
        bool isFinalized;
        uint recordedFees;
        uint totalRewards;
        uint availableRewards;
        mapping(address => uint) unaccountedFeesForAccount;
    }

    address private _periodController;

    /* ========== ADDRESS RESOLVER CONFIGURATION ========== */

    bytes32 private constant CONTRACT_EXCHANGER = "Exchanger";
    bytes32 private constant CONTRACT_SYNTHETIX = "Synthetix";

    bytes32[24] private _addressesToCache = [CONTRACT_EXCHANGER, CONTRACT_SYNTHETIX];

    /* ========== CONSTRUCTOR ========== */

    constructor(
        address owner,
        address periodController,
        address resolver
    ) public Owned(owner) MixinResolver(resolver, _addressesToCache) {
        require(periodController != address(0), "Invalid period controller");

        _periodController = periodController;
    }

    /* ========== VIEWS ========== */

    function synthetix() internal view returns (IERC20) {
        return IERC20(requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing Horizon address"));
    }

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(requireAndGetAddress(CONTRACT_EXCHANGER, "Missing Exchanger address"));
    }

    function getAvailableRewards() external view returns (uint) {
        return _balanceAssignedToRewards;
    }

    function getUnassignedRewards() external view returns (uint) {
        return synthetix().balanceOf(address(this)).sub(_balanceAssignedToRewards);
    }

    function getRewardsToken() external view returns (address) {
        return address(synthetix());
    }

    function getPeriodController() external view returns (address) {
        return _periodController;
    }

    function getCurrentPeriod() external view returns (uint) {
        return _currentPeriodID;
    }

    function getPeriodIsClaimable(uint periodID) external view returns (bool) {
        return _periods[periodID].isFinalized;
    }

    function getPeriodIsFinalized(uint periodID) external view returns (bool) {
        return _periods[periodID].isFinalized;
    }

    function getPeriodRecordedFees(uint periodID) external view returns (uint) {
        return _periods[periodID].recordedFees;
    }

    function getPeriodTotalRewards(uint periodID) external view returns (uint) {
        return _periods[periodID].totalRewards;
    }

    function getPeriodAvailableRewards(uint periodID) external view returns (uint) {
        return _periods[periodID].availableRewards;
    }

    function getUnaccountedFeesForAccountForPeriod(address account, uint periodID) external view returns
(uint) {
        return _periods[periodID].unaccountedFeesForAccount[account];
    }
```

```
    function getAvailableRewardsForAccountForPeriod(address account, uint periodID) external view returns
(uint) {
        return _calculateRewards(account, periodID);
    }

    function getAvailableRewardsForAccountForPeriods(address account, uint[] calldata periodIDs)
        external
        view
        returns (uint totalRewards)
    {
        for (uint i = 0; i < periodIDs.length; i++) {
            uint periodID = periodIDs[i];

            totalRewards = totalRewards.add(_calculateRewards(account, periodID));
        }
    }

    function _calculateRewards(address account, uint periodID) internal view returns (uint) {
        Period storage period = _periods[periodID];
        if (period.availableRewards == 0 || period.recordedFees == 0 || !period.isFinalized) {
            return 0;
        }

        uint accountFees = period.unaccountedFeesForAccount[account];
        if (accountFees == 0) {
            return 0;
        }

        uint participationRatio = accountFees.divideDecimal(period.recordedFees);
        return participationRatio.multiplyDecimal(period.totalRewards);
    }

    /* ========== MUTATIVE FUNCTIONS ========== */

    function claimRewardsForPeriod(uint periodID) external nonReentrant notPaused {
        _claimRewards(msg.sender, periodID);
    }

    function claimRewardsForPeriods(uint[] calldata periodIDs) external nonReentrant notPaused {
        for (uint i = 0; i < periodIDs.length; i++) {
            uint periodID = periodIDs[i];

            // Will revert if any independent claim reverts.
            _claimRewards(msg.sender, periodID);
        }
    }

    function _claimRewards(address account, uint periodID) internal {
        Period storage period = _periods[periodID];
        require(period.isFinalized, "Period is not finalized");

        uint amountToClaim = _calculateRewards(account, periodID);
        require(amountToClaim > 0, "No rewards available");

        period.unaccountedFeesForAccount[account] = 0;
        period.availableRewards = period.availableRewards.sub(amountToClaim);

        _balanceAssignedToRewards = _balanceAssignedToRewards.sub(amountToClaim);

        synthetix().safeTransfer(account, amountToClaim);

        emit RewardsClaimed(account, amountToClaim, periodID);
    }

    /* ========== RESTRICTED FUNCTIONS ========== */

    function recordExchangeFeeForAccount(uint usdFeeAmount, address account) external onlyExchanger {
        Period storage period = _periods[_currentPeriodID];
        // Note: In theory, the current period will never be finalized.
        // Such a require could be added here, but it would just spend gas, since it should always satisfied.

        period.unaccountedFeesForAccount[account]                                                    =
period.unaccountedFeesForAccount[account].add(usdFeeAmount);
        period.recordedFees = period.recordedFees.add(usdFeeAmount);

        emit ExchangeFeeRecorded(account, usdFeeAmount, _currentPeriodID);
    }

    function closeCurrentPeriodWithRewards(uint rewards) external onlyPeriodController {
        uint currentBalance = synthetix().balanceOf(address(this));
        uint availableForNewRewards = currentBalance.sub(_balanceAssignedToRewards);
        require(rewards <= availableForNewRewards, "Insufficient free rewards");

        Period storage period = _periods[_currentPeriodID];

        period.totalRewards = rewards;
        period.availableRewards = rewards;
```

```
        period.isFinalized = true;

        _balanceAssignedToRewards = _balanceAssignedToRewards.add(rewards);

        emit PeriodFinalizedWithRewards(_currentPeriodID, rewards);

        _currentPeriodID = _currentPeriodID.add(1);

        emit NewPeriodStarted(_currentPeriodID);
    }

    function recoverTokens(address tokenAddress, address recoverAddress) external onlyOwner {
        _validateRecoverAddress(recoverAddress);
        require(tokenAddress != address(synthetix()), "Must use another function");

        IERC20 token = IERC20(tokenAddress);

        uint tokenBalance = token.balanceOf(address(this));
        require(tokenBalance > 0, "No tokens to recover");

        token.safeTransfer(recoverAddress, tokenBalance);

        emit TokensRecovered(tokenAddress, recoverAddress, tokenBalance);
    }

    function recoverUnassignedRewardTokens(address recoverAddress) external onlyOwner {
        _validateRecoverAddress(recoverAddress);

        uint tokenBalance = synthetix().balanceOf(address(this));
        require(tokenBalance > 0, "No tokens to recover");

        uint unassignedBalance = tokenBalance.sub(_balanceAssignedToRewards);
        require(unassignedBalance > 0, "No tokens to recover");

        synthetix().safeTransfer(recoverAddress, unassignedBalance);

        emit UnassignedRewardTokensRecovered(recoverAddress, unassignedBalance);
    }

    function recoverAssignedRewardTokensAndDestroyPeriod(address recoverAddress, uint periodID) external
onlyOwner {
        _validateRecoverAddress(recoverAddress);
        require(periodID < _currentPeriodID, "Cannot recover from active");

        Period storage period = _periods[periodID];
        require(period.availableRewards > 0, "No rewards available to recover");

        uint amount = period.availableRewards;
        synthetix().safeTransfer(recoverAddress, amount);

        _balanceAssignedToRewards = _balanceAssignedToRewards.sub(amount);

        delete _periods[periodID];

        emit AssignedRewardTokensRecovered(recoverAddress, amount, periodID);
    }

    function _validateRecoverAddress(address recoverAddress) internal view {
        if (recoverAddress == address(0) || recoverAddress == address(this)) {
            revert("Invalid recover address");
        }
    }

    function setPeriodController(address newPeriodController) external onlyOwner {
        require(newPeriodController != address(0), "Invalid period controller");

        _periodController = newPeriodController;

        emit PeriodControllerChanged(newPeriodController);
    }

    /* ========== MODIFIERS ========== */

    modifier onlyPeriodController() {
        require(msg.sender == _periodController, "Caller not period controller");
        _;
    }

    modifier onlyExchanger() {
        require(msg.sender == address(exchanger()), "Only Exchanger can invoke this");
        _;
    }

    /* ========== EVENTS ========== */

    event ExchangeFeeRecorded(address indexed account, uint amount, uint periodID);
    event RewardsClaimed(address indexed account, uint amount, uint periodID);
```

```
    event NewPeriodStarted(uint periodID);
    event PeriodFinalizedWithRewards(uint periodID, uint rewards);
    event TokensRecovered(address tokenAddress, address recoverAddress, uint amount);
    event UnassignedRewardTokensRecovered(address recoverAddress, uint amount);
    event AssignedRewardTokensRecovered(address recoverAddress, uint amount, uint periodID);
    event PeriodControllerChanged(address newPeriodController);
}
```

VirtualSynth.sol

```
pragma solidity ^0.5.16;

// Inheritance
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/ERC20.sol";

// Libraries
import "./SafeDecimalMath.sol";

// Internal references
import "./interfaces/ISynth.sol";
import "./interfaces/IAddressResolver.sol";
import "./interfaces/IVirtualSynth.sol";
import "./interfaces/IExchanger.sol";
// Note: use OZ's IERC20 here as using ours will complain about conflicting names
// during the build
import "openzeppelin-solidity-2.3.0/contracts/token/ERC20/IERC20.sol";


// https://docs.synthetix.io/contracts/source/contracts/virtualsynth
contract VirtualSynth is ERC20, IVirtualSynth {
    using SafeMath for uint;
    using SafeDecimalMath for uint;

    IERC20 public synth;
    IAddressResolver public resolver;

    bool public settled = false;

    uint8 public constant decimals = 18;

    // track initial supply so we can calculate the rate even after all supply is burned
    uint public initialSupply;

    // track final settled amount of the synth so we can calculate the rate after settlement
    uint public settledAmount;

    bytes32 public currencyKey;

    constructor(
        IERC20 _synth,
        IAddressResolver _resolver,
        address _recipient,
        uint _amount,
        bytes32 _currencyKey
    ) public ERC20() {
        synth = _synth;
        resolver = _resolver;
        currencyKey = _currencyKey;

        // Assumption: the synth will be issued to us within the same transaction,
        // and this supply matches that
        _mint(_recipient, _amount);

        initialSupply = _amount;
    }

    // INTERNALS

    function exchanger() internal view returns (IExchanger) {
        return IExchanger(resolver.requireAndGetAddress("Exchanger", "Exchanger contract not found"));
    }

    function secsLeft() internal view returns (uint) {
        return exchanger().maxSecsLeftInWaitingPeriod(address(this), currencyKey);
    }

    function calcRate() internal view returns (uint) {
        if (initialSupply == 0) {
            return 0;
        }

        uint synthBalance;

        if (!settled) {
            synthBalance = IERC20(address(synth)).balanceOf(address(this));
            (uint reclaim, uint rebate, ) = exchanger().settlementOwing(address(this), currencyKey);
```

```
                if (reclaim > 0) {
                    synthBalance = synthBalance.sub(reclaim);
                } else if (rebate > 0) {
                    synthBalance = synthBalance.add(rebate);
                }
            } else {
                synthBalance = settledAmount;
            }

            return synthBalance.divideDecimalRound(initialSupply);
        }

        function balanceUnderlying(address account) internal view returns (uint) {
            uint vBalanceOfAccount = balanceOf(account);

            return vBalanceOfAccount.multiplyDecimalRound(calcRate());
        }

        function settleSynth() internal {
            if (settled) {
                return;
            }
            settled = true;

            exchanger().settle(address(this), currencyKey);

            settledAmount = IERC20(address(synth)).balanceOf(address(this));

            emit Settled(totalSupply(), settledAmount);
        }

        // VIEWS

        function name() external view returns (string memory) {
            return string(abi.encodePacked("Virtual Zasset ", currencyKey));
        }

        function symbol() external view returns (string memory) {
            return string(abi.encodePacked("v", currencyKey));
        }

        // get the rate of the vSynth to the synth.
        function rate() external view returns (uint) {
            return calcRate();
        }

        // show the balance of the underlying synth that the given address has, given
        // their proportion of totalSupply
        function balanceOfUnderlying(address account) external view returns (uint) {
            return balanceUnderlying(account);
        }

        function secsLeftInWaitingPeriod() external view returns (uint) {
            return secsLeft();
        }

        function readyToSettle() external view returns (bool) {
            return secsLeft() == 0;
        }

        // PUBLIC FUNCTIONS

        // Perform settlement of the underlying exchange if required,
        // then burn the accounts vSynths and transfer them their owed balanceOfUnderlying
        function settle(address account) external {
            settleSynth();

            IERC20(address(synth)).transfer(account, balanceUnderlying(account));

            _burn(account, balanceOf(account));
        }

        event Settled(uint totalSupply, uint amountAfterSettled);
}
```

BinaryOptionMarketData.sol

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

interface IERC20 {
    // ERC20 Optional Views
    function name() external view returns (string memory);

    function symbol() external view returns (string memory);
```

```
    function decimals() external view returns (uint8);

    // Views
    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    // Mutative functions
    function transfer(address to, uint value) external returns (bool);

    function approve(address spender, uint value) external returns (bool);

    function transferFrom(
        address from,
        address to,
        uint value
    ) external returns (bool);

    // Events
    event Transfer(address indexed from, address indexed to, uint value);

    event Approval(address indexed owner, address indexed spender, uint value);
}

interface IBinaryOptionMarketManager {
    /* ========== TYPES ========== */

    struct Fees {
        uint poolFee;
        uint creatorFee;
        uint refundFee;
    }

    struct Durations {
        uint maxOraclePriceAge;
        uint expiryDuration;
        uint maxTimeToMaturity;
    }

    struct CreatorLimits {
        uint capitalRequirement;
        uint skewLimit;
    }

    /* ========== VIEWS / VARIABLES ========== */

    function fees() external view returns (uint poolFee, uint creatorFee, uint refundFee);
    function durations() external view returns (uint maxOraclePriceAge, uint expiryDuration, uint
  maxTimeToMaturity);
    function creatorLimits() external view returns (uint capitalRequirement, uint skewLimit);

    function marketCreationEnabled() external view returns (bool);
    function totalDeposited() external view returns (uint);

    function numActiveMarkets() external view returns (uint);
    function activeMarkets(uint index, uint pageSize) external view returns (address[] memory);
    function numMaturedMarkets() external view returns (uint);
    function maturedMarkets(uint index, uint pageSize) external view returns (address[] memory);

    /* ========== MUTATIVE FUNCTIONS ========== */

    function createMarket(
        bytes32 oracleKey, uint strikePrice,
        uint[2] calldata times, // [biddingEnd, maturity]
        uint[2] calldata bids // [longBid, shortBid]
    ) external returns (IBinaryOptionMarket);
    function resolveMarket(address market) external;
    function expireMarkets(address[] calldata market) external;
}


interface IBinaryOptionMarket {
    /* ========== TYPES ========== */

    enum Phase { Bidding, Trading, Maturity, Expiry }
    enum Side { Long, Short }

    struct Options {
        IBinaryOption long;
        IBinaryOption short;
    }

    struct Prices {
```

```
        uint long;
        uint short;
    }

    struct Times {
        uint biddingEnd;
        uint maturity;
        uint expiry;
    }

    struct OracleDetails {
        bytes32 key;
        uint strikePrice;
        uint finalPrice;
    }

    /* ========== VIEWS / VARIABLES ========== */

    function options() external view returns (IBinaryOption long, IBinaryOption short);
    function prices() external view returns (uint long, uint short);
    function times() external view returns (uint biddingEnd, uint maturity, uint destructino);
    function oracleDetails() external view returns (bytes32 key, uint strikePrice, uint finalPrice);
    function fees() external view returns (uint poolFee, uint creatorFee, uint refundFee);
    function creatorLimits() external view returns (uint capitalRequirement, uint skewLimit);

    function deposited() external view returns (uint);
    function creator() external view returns (address);
    function resolved() external view returns (bool);

    function phase() external view returns (Phase);
    function oraclePriceAndTimestamp() external view returns (uint price, uint updatedAt);
    function canResolve() external view returns (bool);
    function result() external view returns (Side);

    function pricesAfterBidOrRefund(Side side, uint value, bool refund) external view returns (uint long, uint short);
    function bidOrRefundForPrice(Side bidSide, Side priceSide, uint price, bool refund) external view returns (uint);

    function bidsOf(address account) external view returns (uint long, uint short);
    function totalBids() external view returns (uint long, uint short);
    function claimableBalancesOf(address account) external view returns (uint long, uint short);
    function totalClaimableSupplies() external view returns (uint long, uint short);
    function balancesOf(address account) external view returns (uint long, uint short);
    function totalSupplies() external view returns (uint long, uint short);
    function exercisableDeposits() external view returns (uint);

    /* ========== MUTATIVE FUNCTIONS ========== */

    function bid(Side side, uint value) external;
    function refund(Side side, uint value) external returns (uint refundMinusFee);

    function claimOptions() external returns (uint longClaimed, uint shortClaimed);
    function exerciseOptions() external returns (uint);
}



interface IBinaryOption {
    /* ========== VIEWS / VARIABLES ========== */

    function market() external view returns (IBinaryOptionMarket);

    function bidOf(address account) external view returns (uint);
    function totalBids() external view returns (uint);

    function balanceOf(address account) external view returns (uint);
    function totalSupply() external view returns (uint);

    function claimableBalanceOf(address account) external view returns (uint);
    function totalClaimableSupply() external view returns (uint);
}


contract BinaryOptionMarketData {


    struct OptionValues {
        uint long;
        uint short;
    }

    struct Deposits {
        uint deposited;
        uint exercisableDeposits;
```

```
    }

    struct Resolution {
        bool resolved;
        bool canResolve;
    }

    struct OraclePriceAndTimestamp {
        uint price;
        uint updatedAt;
    }

    // used for things that don't change over the lifetime of the contract
    struct MarketParameters {
        address creator;
        IBinaryOptionMarket.Options options;
        IBinaryOptionMarket.Times times;
        IBinaryOptionMarket.OracleDetails oracleDetails;
        IBinaryOptionMarketManager.Fees fees;
        IBinaryOptionMarketManager.CreatorLimits creatorLimits;
    }

    struct MarketData {
        OraclePriceAndTimestamp oraclePriceAndTimestamp;
        IBinaryOptionMarket.Prices prices;
        Deposits deposits;
        Resolution resolution;
        IBinaryOptionMarket.Phase phase;
        IBinaryOptionMarket.Side result;
        OptionValues totalBids;
        OptionValues totalClaimableSupplies;
        OptionValues totalSupplies;
    }

    struct AccountData {
        OptionValues bids;
        OptionValues claimable;
        OptionValues balances;
    }

    function getMarketParameters(IBinaryOptionMarket market) public view returns (MarketParameters memory) {

        (IBinaryOption long, IBinaryOption short) = market.options();
        (uint biddingEndDate, uint maturityDate, uint expiryDate) = market.times();
        (bytes32 key, uint strikePrice, uint finalPrice) = market.oracleDetails();
        (uint poolFee, uint creatorFee, uint refundFee) = market.fees();

        MarketParameters memory data = MarketParameters(
            market.creator(),
            IBinaryOptionMarket.Options(long, short),
            IBinaryOptionMarket.Times(biddingEndDate,maturityDate,expiryDate),
            IBinaryOptionMarket.OracleDetails(key, strikePrice, finalPrice),
            IBinaryOptionMarketManager.Fees(poolFee, creatorFee, refundFee),
            IBinaryOptionMarketManager.CreatorLimits(0, 0)
        );

        // Stack too deep otherwise.
        (uint capitalRequirement, uint skewLimit) = market.creatorLimits();
        data.creatorLimits = IBinaryOptionMarketManager.CreatorLimits(capitalRequirement, skewLimit);
        return data;
    }

    function getMarketData(IBinaryOptionMarket market) public view returns (MarketData memory) {

        (uint price, uint updatedAt) = market.oraclePriceAndTimestamp();
        (uint longClaimable, uint shortClaimable) = market.totalClaimableSupplies();
        (uint longSupply, uint shortSupply) = market.totalSupplies();
        (uint longBids, uint shortBids) = market.totalBids();
        (uint longPrice, uint shortPrice) = market.prices();

        return MarketData(
            OraclePriceAndTimestamp(price, updatedAt),
            IBinaryOptionMarket.Prices(longPrice, shortPrice),
            Deposits(market.deposited(), market.exercisableDeposits()),
            Resolution(market.resolved(), market.canResolve()),
            market.phase(),
            market.result(),
            OptionValues(longBids, shortBids),
            OptionValues(longClaimable, shortClaimable),
            OptionValues(longSupply, shortSupply)
        );
    }

    function getAccountMarketData(IBinaryOptionMarket market, address account) public view returns (AccountData memory) {
        (uint longBid, uint shortBid) = market.bidsOf(account);
```

```
        (uint longClaimable, uint shortClaimable) = market.claimableBalancesOf(account);
        (uint longBalance, uint shortBalance) = market.balancesOf(account);

        return AccountData(
            OptionValues(longBid, shortBid),
            OptionValues(longClaimable, shortClaimable),
            OptionValues(longBalance, shortBalance)
        );
    }
}
```

SynthSummaryUtil.sol

```
pragma solidity ^0.5.16;

interface ISynth {
    function currencyKey() external view returns (bytes32);
    function balanceOf(address owner) external view returns (uint);
    function totalSupply() external view returns (uint);
}
interface ISynthetix {
    function availableSynths(uint index) external view returns (ISynth);
    function availableSynthCount() external view returns (uint);
    function availableCurrencyKeys() external view returns (bytes32[] memory);
}

interface IExchangeRates {
    function rateIsFrozen(bytes32 currencyKey) external view returns (bool);
    function ratesForCurrencies(bytes32[] calldata currencyKeys) external view returns (uint[] memory);
    function effectiveValue(bytes32 sourceCurrencyKey, uint sourceAmount, bytes32 destinationCurrencyKey)
    external
    view
    returns (uint);
}

interface IAddressResolver {
    function getAddress(bytes32 name) external view returns (address);
    function getSynth(bytes32 key) external view returns (address);
    function requireAndGetAddress(bytes32 name, string calldata reason) external view returns (address);
}

contract SynthSummaryUtil {

    IAddressResolver public addressResolverProxy;

    bytes32 internal constant CONTRACT_SYNTHETIX = "Synthetix";
    bytes32 internal constant CONTRACT_EXRATES = "ExchangeRates";
    bytes32 internal constant SUSD = "zUSD";

    constructor(address resolver) public {
        addressResolverProxy = IAddressResolver(resolver);
    }

    function _synthetix() internal view returns (ISynthetix) {
        return ISynthetix(addressResolverProxy.requireAndGetAddress(CONTRACT_SYNTHETIX, "Missing
Horizon address"));
    }

    function _exchangeRates() internal view returns (IExchangeRates) {
        return IExchangeRates(addressResolverProxy.requireAndGetAddress(CONTRACT_EXRATES, "Missing
ExchangeRates address"));
    }

    function totalSynthsInKey(address account, bytes32 currencyKey) external view returns (uint total) {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            total += exchangeRates.effectiveValue(synth.currencyKey(), synth.balanceOf(account),
currencyKey);
        }
        return total;
    }

    function synthsBalances(address account) external view returns (bytes32[] memory, uint[] memory, uint[]
memory) {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        bytes32[] memory currencyKeys = new bytes32[](numSynths);
        uint[] memory balances = new uint[](numSynths);
        uint[] memory sUSDBalances = new uint[](numSynths);
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            currencyKeys[i] = synth.currencyKey();
```

```
            balances[i] = synth.balanceOf(account);
            sUSDBalances[i] = exchangeRates.effectiveValue(currencyKeys[i], balances[i], SUSD);
        }
        return (currencyKeys, balances, sUSDBalances);
    }

    function frozenSynths() external view returns (bytes32[] memory) {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();
        uint numSynths = synthetix.availableSynthCount();
        bytes32[] memory frozenSynthsKeys = new bytes32[](numSynths);
        for (uint i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            if (exchangeRates.rateIsFrozen(synth.currencyKey())) {
                frozenSynthsKeys[i] = synth.currencyKey();
            }

        }
        return frozenSynthsKeys;
    }

    function synthsRates() external view returns (bytes32[] memory, uint[] memory) {
        bytes32[] memory currencyKeys = _synthetix().availableCurrencyKeys();
        return (currencyKeys, _exchangeRates().ratesForCurrencies(currencyKeys));
    }

    function synthsTotalSupplies()
        external
        view
        returns (bytes32[] memory, uint256[] memory, uint256[] memory)
    {
        ISynthetix synthetix = _synthetix();
        IExchangeRates exchangeRates = _exchangeRates();

        uint256 numSynths = synthetix.availableSynthCount();
        bytes32[] memory currencyKeys = new bytes32[](numSynths);
        uint256[] memory balances = new uint256[](numSynths);
        uint256[] memory sUSDBalances = new uint256[](numSynths);
        for (uint256 i = 0; i < numSynths; i++) {
            ISynth synth = synthetix.availableSynths(i);
            currencyKeys[i] = synth.currencyKey();
            balances[i] = synth.totalSupply();
            sUSDBalances[i] = exchangeRates.effectiveValue(
                currencyKeys[i],
                balances[i],
                SUSD
            );
        }
        return (currencyKeys, balances, sUSDBalances);
    }
}
```

6. Appendix B: Vulnerability rating standard

Smart contract vulnerability rating standards	
Level	Level Description
High	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose BNB or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending BNB to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.
Medium	High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.
Low	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of BNB or tokens to trigger, vulnerabilities where attackers cannot

	directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk Wait.

7. Appendix C：Introduction to auditing tools

7.1 Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

7.2 Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

7.3 securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a

specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

7.4 Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

7.5 MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

7.6 ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

7.7 ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

7.8 Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

7.9 Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone	+86(10)400 060 9587
E-mail	sec@knownsec.com
Website	www.knownsec.com
Address	wangjing soho T2-B2509,Chaoyang District, Beijing