SFT Protocol Documentation

Benjamin Hauser, Alex Firmani

Feb 10, 2019

Contents

The Secured Financial Transaction Protocol (SFT) is a set of compliance-oriented smart contracts, written in Solidity for the Ethereum blockchain, that allow for the tokenization of debt and equity based securities. It provides a robust, flexible framework allowing issuers and investors to retain regulatory compliance throughout primary issuance and multi-jurisdictional secondary trading.

The SFT Protocol was developed by HyperLink Technology.

CHAPTER 1

How it Works

SFT is designed to maximize interoperability between different network participants. Broadly speaking, these participants may be split into four categories:

- **Investors** are entities that have passed KYC/AML checks and are are able to hold or transfer security tokens.
- **Issuers** are entities that create and sell security tokens to fund their business operations.
- **Registrars** are trusted entities that provide KYC/AML services for network participants.
- **Custodians** are trusted entities that may hold tokens on behalf of multiple investors and facilitate secondary trading of tokens.

The protocol is built with two central concepts in mind: **identification** and **permission**. Each investor has their identity verified by a registrar and a unique ID hash is associated to their wallet addresses. Based on this identity information, issuers and custodians apply a series of rules to determine how the investor may interact with them.

Issuers, registrars and custodians each exist on the blockchain with their own smart contracts that define the way they interact with one another. These contracts allow different entities to provide services to each other within the ecosystem.

Security tokens in the protocol are built upon the ERC20 token standard. Tokens are transferred via the `transfer` and `transferFrom` methods, however the transfer will only succeed if it passes a series of on-chain compliance requirements. A call to `checkTransfer` returns true if the transfer is possible. The base configuration includes investor identification, tracking investor counts and limits, and restrictions on countries and accredited status. By implementing other modules a variety of additional functionality is possible so as to allow compliance to laws in the countries of the issuer and investors.

CHAPTER 2

Components

The SFT protocol is comprised of four core contracts:

1. *SecurityToken*
    - ERC20 compliant token contract
    - Intended to represent a claim to ownership of securities
    - Permissioning logic to enforce compliance in all token transfers
    - Modular design allows for optional added functionality
2. *IssuingEntity*
    - Owner contract for tokens created by the same issuer
    - Handles common compliance logic for all the issuer's tokens
    - Modular design allows for optional added functionality
    - Multi-sig, multi-authority design provides increased security and permissioned contract management
3. *KYCRegistrar*
    - Whitelists that provide identity, region, and accreditation information of investors based on off-chain KYC/AML verification
    - May be maintained by a single entity or a federation across multiple jurisdictions
    - Multi-sig, multi-authority design provides increased security and permissioned contract management
4. *Custodian*
    - Contracts that represent an entity approved to hold tokens on behalf of multiple investors
    - Interacts with IssuingEntity to provide accurate on-chain investor counts
    - Intended to be used by broker/dealers and exchanges
    - Modular design allows for optional added functionality
    - Multi-sig, multi-authority design provides increased security and permissioned contract management

CHAPTER 3

Yellow Paper

The Yellow Paper provides a more detailed overview of how the protocol is structrued.

CHAPTER 4

Source Code

The SFT Protocol is open source. You can view the code on GitHub.

CHAPTER 5

Testing and Deployment

Unit testing and deployment of this project is performed with Brownie.

Warning: The SFT Protocol is still under active development and has not yet undergone a third party audit. Please notify us if you find any issues in the code. We highly recommend against using these contracts prior to an audit by a trusted third party.

CHAPTER 6

License

This project is licensed under the Apache 2.0 license.

CHAPTER 7

Contents

Keyword Index, *Glossary*

7.1 Getting Started

This is a quick explanation of the minimum steps required to deploy and use each contract of the protocol.

To setup a simple test environment using brownie:

```
brownie console
>>> run('simple')
```

This runs simple.py which:

- Deploys `KYCRegistrar` from `accounts[0]`
- Deploys `IssuingEntity` from `accounts[1]`
- Deploys `SecurityToken` from `accounts[1]` with an initial total supply of 1,000,000 tokens
- Associates the contracts
- Approves `accounts[2:8]` in `KYCRegistrar`, with investor ratings 1-2 and country codes 1-3
- Approves investors from country codes 1-3 in `IssuingEntity`

From this configuration, the contracts are ready to transfer tokens:

```
>>> SecurityToken[0].transfer(accounts[2], 1000)
>>> SecurityToken[0].transfer(accounts[3], 1000, {'from': accounts[2]})
```

7.1.1 KYC Registrar

To setup an investor registry, deploy KYCRegistrar.sol. Owner addresses will then be able to add investors using `KYCRegistrar.addInvestor` or approve other whitelisting authorities with `KYCRegistrar.`

`addAuthority`.

See the *KYCRegistrar* page for a detailed explanation of how to use this contract.

7.1.2 Issuing Tokens

Issuing tokens and being able to transfer them requires the following steps:

1. Deploy IssuingEntity.sol.
2. Call `IssuingEntity.setRegistrar` to add one or more investor registries. You may maintain your own registry and/or use those belonging to trusted third parties.
3. Deploy SecurityToken.sol. Enter the address of the issuer contract from step 1 in the constructor. The total supply of tokens will be initially creditted to the issuer.
4. Call `IssuingEntity.addToken` to attach the token to the issuer.
5. Call `IssuingEntity.setCountries` to approve investors from specific countries to hold the tokens.

At this point, the issuer will be able to transfer tokens to any address that has been whitelisted by one of the approved investor registries *if the investor meets the country and rating requirements*.

Note that the issuer's balance is assigned to the IssuingEntity contract. The issuer can transfer these tokens with a normal call to `SecurityToken.transfer` from any approved address. Sending tokens to any address associated with the issuer will increase the balance on the IssuingEntity contract.

See the *IssuingEntity* and *SecurityToken* pages for detailed explanations of how to use these contracts.

7.1.3 Transferring Tokens

SecurityToken.sol is based on the ERC20 Token Standard. Token transfers may be performed in the same ways as any token using this standard. However, in order to send or receive tokens you must:

- Be approved in one of the KYC registries associated to the token issuer
- Meet the approved country and rating requirements as set by the issuer
- Pass any additional checks set by the issuer

You can check if a transfer will succeed without performing a transaction by calling the `SecurityToken.checkTransfer` method within the token contract.

Restrictions imposed on investor limits, approved countries and minimum ratings are only checked when receiving tokens. Unless an address has been explicitly blocked, it will always be able to send an existing balance. For example, an investor may purchase tokens that are only available to accredited investors, and then later their accreditation status expires. The investor may still transfer the tokens they already have, but may not receive any more tokens.

Transferring a balance between two addresses associated with the same investor ID does not have the same restrictions imposed, as there is no change of ownership. An investor with multiple addresses may call `SecurityToken.transferFrom` to move tokens from any of their addresses without first using the `SecurityToken.approve` method. The issuer can also use `SecurityToken.transferFrom` to move any investor's tokens, without prior approval.

See the *SecurityToken* page for a detailed explanation of how to use this contract.

7.1.4 Custodians

To set up a custodian contract to send and receive tokens, deploy Custodian.sol and then attach it to an IssuingEntity with `IssuingEntity.addCustodian`. At this point, investors may send tokens into the custodian contract just like they would any other address.

The `Custodian.transfer` function allows you to send tokens out of the contract. You may modify the list of beneficial owners using `addInvestors` and `removeInvestors`.

See the *Custodian* page for a detailed explanation of how to use this contract.

7.2 SecurityToken

Each SecurityToken contract represents a single, fungible class of securities from an issuer. The contracts conforms to the ERC20 Token Standard, with an additional `checkTransfer` function available to verify if a transfer will succeed.

Token contracts are associated to an *IssuingEntity* and also implement *Modules* functionality. Permissioning around transfers is achieved through these components. See the respective documents for more detailed information.

This documentation only explains contract methods that are meant to be accessed directly. External methods that will revert unless called through another contract, such as IssuingEntity or modules, are not included.

It may be useful to also view the SecurityToken.sol source code while reading this document.

7.2.1 Deployment

The constructor takes the following arguments:

`SecurityToken.`**`constructor`**`(`*address _issuer*, *string _name*, *string _symbol*, *uint256 _totalSupply*`)`

- `_issuer`: The address of the `IssuingEntity` associated with this token.
- `_name`: The full name of the token.
- `_symbol`: The ticker symbol for the token.
- `_totalSupply`: The initial total supply of tokens to create.

The total supply of tokens is assigned to the issuer at the time of creation, with a `Transfer` event logged to show them as moving from 0x00.

After the contract is deployed it must be associated with the issuer via `IssuingEntity.addToken`. Token transfers are not possible until this is done.

7.2.2 Constants

The following public variables cannot be changed after contract deployment.

`SecurityToken.`**`name`**`()`
The full name of the security token.

`SecurityToken.`**`symbol`**`()`
The ticker symbol for the token.

`SecurityToken.`**`decimals`**`()`
The number of decimal places for the token. This is a constant always set to 0, as you cannot legally fractionalize a security.

`SecurityToken.`**`ownerID`**`()`
The bytes32 ID hash of the issuer associated with this token.

`SecurityToken.`**`issuer`**`()`
The address of the associated IssuingEntity contract.

7.2.3 Total Supply and Balances

Along with the standard ERC20 methods, SecurityToken introduces two additional methods around the total supply.

`SecurityToken.`**`totalSupply`**`()`
Returns the total supply of tokens.

`SecurityToken.`**`balanceOf`**`(`*address*`)`
Returns the token balance for a given address.

`SecurityToken.`**`treasurySupply`**`()`
Returns the number of tokens held by the issuer. Equivalent to calling `SecurityToken.balanceOf(SecurityToken.ownerID())`.

`SecurityToken.`**`circulatingSupply`**`()`
Returns the total supply, less the amount held by the issuer.

7.2.4 Token Transfers

SecurityToken uses the standard ERC20 methods for token transfers, however their functionality differs slightly due to transfer permissioning requirements.

`SecurityToken.`**`checkTransfer`**`(`*address _from*, *address _to*, *uint256 _value*`)`
Returns true if `_from` is permitted to transfer `_value` tokens to `_to`.

For a transfer to succeed it must first pass a series of checks:

- Tokens cannot be locked.
- Sender must have a sufficient balance.
- Sender and receiver must be verified in a registrar associated to the issuer.
- Sender and receiver must not be restricted by the registrar or the issuer.
- Transfer must not result in any issuer-imposed investor limits being exceeded.
- Transfer must be permitted by all active modules.

Transfers between two addresses that are associated to the same ID do not undergo the same level of restrictions, as there is no change of ownership occuring.

`SecurityToken.`**`transfer`**`(`*address _to*, *uint256 _value*`)`
Transfers `_value` tokens from `msg.sender` to `_to`.

All transfers will log the `Transfer` event. Transfers where there is a change of ownership will also log‘‘IssuingEntity.TransferOwnership‘‘.

`SecurityToken.`**`approve`**`(`*address _spender*, *uint256 _value*`)`
Approves `_spender` to transfer up to `_value` tokens belonging to `msg.sender`.

Approval may be given to any address, but a transfer can only be initiated by an address that is known by one of the associated registrars. The same transfer checks also apply for both the sender and receiver, as if the transfer was done directly.

`SecurityToken.`**`transferFrom`**`(`*address _from*, *address _to*, *uint256 _value*`)`

Transfers `_value` tokens from `_from` to `_to`.

If the caller and sender addresses are both associated to the same ID, `transferFrom` may be called without giving prior approval. In this way an investor can easily recover tokens when a private key is lost or compromised.

7.2.5 Issuer Balances and Transfers

Tokens held by the issuer will always be at the address of the IssuingEntity contract. `SecurityToken.treasurySupply()` will return the same result as `SecurityToken.balanceOf(SecurityToken.issuer())`.

As a result, the following non-standard behaviours exist:

- Any address associated with the issuer can transfer tokens from the IssuingEntity contract using `SecurityToken.transfer`.
- Attempting to send tokens to any address associated with the issuer will result in the tokens being sent to the IssuingEntity contract.

The issuer may call `SecurityToken.transferFrom` to move tokens between any addresses without prior approval. Transfers of this type must still pass the normal checks, with the exception that the sending address may be restricted. In this way the issuer can aid investors with token recovery in the event of a lost or compromised private key, or force a transfer in the event of a court order or sanction.

7.2.6 Modules

Modules are attached and detached via *IssuingEntity*.

`SecurityToken.`**`isActiveModule`**`(`*address _module*`)`

Returns true if a module is currently active on the token. Modules that are active on the IssuingEntity are also considered active on tokens.

7.3 IssuingEntity

IssuingEntity contracts hold shared compliance logic for all security tokens created by a single issuer. They are the central contract that an issuer uses to connect and interact with registrars, tokens and custodians.

Each issuer contract includes standard SFT protocol *MultiSig Implementation* and *Modules* functionality. See the respective documents for detailed information on these components.

This documentation only explains contract methods that are meant to be accessed directly. External methods that will revert unless called through another contract, such as a token or module, are not included.

It may be useful to also view the IsssuingEntity.sol source code while reading this document.

7.3.1 Deployment

The constructor declares the owner as per standard *MultiSig Implementation*.

`IssuingEntity.`**`constructor`**`(`*address[] _owners*, *uint32 _threshold*`)`

- `_owners`: One or more addresses to associate with the contract owner. The address deploying the contract is not implicitly included within the owner list.

- `_threshold`: The number of calls required for the owner to perform a multi-sig action.

The ID of the owner is generated as a keccak of the contract address and available from the public getter `ownerID`.

7.3.2 Constants

The following public variables cannot be changed after contract deployment.

`SecurityToken.`**`ownerID`**`()`

The bytes32 ID hash of the issuer.

7.3.3 Adding and Restricting Tokens

Tokens must be associated with the IssuingEntity contract before they can be transfered.

`IssuingEntity.`**`addToken`**`(`*address _token*`)`

Associates a *SecurityToken* contract with the IssuingEntity.

`IssuingEntity.`**`setTokenRestriction`**`(`*address _token*, *bool _allowed*`)`

Restricts or unrestricts transfers of a token. When a token is restricted, only the issuer may perform transfers.

`IssuingEntity.`**`setGlobalRestriction`**`(`*bool _allowed*`)`

Restricts or unrestricts transfers of all associated tokens. Modifying the global restriction does not affect individual token restrictions.

7.3.4 Identifying Investors

Investors must be identified by a registrar before they can send or receive tokens. This identity data is then used to apply further checks against investor limits and accreditation requirements.

`IssuingEntity.`**`setRegistrar`**`(`*address _registrar*, *bool _allowed*`)`

Associates or removes a *KYCRegistrar*.

Before a transfer is completed, each associated registrar is called to check which IDs are associated to the transfer addresses.

The address => ID association is stored within IssuingEntity. If a registrar is later removed it is impossible for another registrar to return a different ID for the address.

When a registrar is removed, any investors that were identified through it will be unable to send or receive tokens until they are identified through another associated registrar. Transfer attempts will revert with the message "Registrar restricted".

`IssuingEntity.`**`getID`**`(`*address _addr*`)`

Returns the investor ID associated with an address. If the address is not saved in the contract, this call will query associated registrars.

`IssuingEntity.`**`getInvestorRegistrar`**`(`*bytes32 _id*`)`

Returns the registrar address associated with an investor ID. If the investor ID is not saved in the contract, this call will return 0x00.

`IssuingEntity.`**`setInvestorRestriction`**`(`*bytes32 _id*, *bool _allowed*`)`

Retricts or permits an investor from transferring tokens, based on their ID.

This can only be used to block an investor that would otherwise be able to hold the tokens, it cannot be used to whitelist investors who are not listed in an associated registrar. When an investor is restricted, the issuer is still able to transfer tokens from their addresses.

7.3.5 Custodians

Custodian are entities that are approved to hold tokens on behalf of multiple investors. Common examples of custodians include broker/dealers, escrow agents and secondary markets. Each custodian must be individually approved by an issuer before they can receive tokens.

Custodians interact with an issuer's investor counts differently from regular investors. When an investor transfers a balance into a custodian it does not increase the overall investor count, instead the investor is now included in the list of beneficial owners represented by the custodian. Even if the investor now has a balance of 0, they will be still be included in the issuer's investor count.

Each time a beneficial owner is added or removed from a custodian, the `BeneficialOwnerSet` event will fire. Filtering for this event can be used to keep an up-to-date record of which investors have tokens held by a custodian.

See the *Custodian* documentation for more information on how custodians interact with the IssuingEntity contract.

`IssuingEntity.`**`addCustodian`**(*address _custodian*)

Approves a custodian contract to send and receive tokens associated with the issuer.

Once a custodian is approved, they can be restricted with `IssuingEntity.setInvestorRestriction`.

`IssuingEntity.`**`releaseOwnership`**(*bytes32 _custID*, *bytes32 _id*)

Removes an investor from a custodian's list of beneficial owners.

- `_custID`: Custodian ID
- `_id`: Investor ID

This can be called via the Custodian contract, or directly by the issuer.

Note: In the case of a direct call by the issuer, the Custodian contract will not be called to update it's record. This results in a discrepancy between the on-chain ownership records of the custodian contract and the issuer contract. An issuer should only call this method as a last resort in a situation where a custodian has been found to be acting in bad faith.

7.3.6 Setting Investor Limits

Issuers can define investor limits globally, by country, by investor rating, or by a combination thereof. These limits are shared across all tokens associated to the issuer.

Investor counts and limits are stored in uint32[8] arrays. The first entry in each array is the sum of all the remaining entries. The remaining entries correspond to the count or limit for each investor rating. In most (if not all) countries there will be less than 7 types of investor accreditation ratings, and so the upper range of these arrays will be empty. Setting an investor limit to 0 means no limit is imposed.

The issuer must explicitely approve each country from which investors are allowed to purchase tokens.

It is possible for an issuer to set a limit that is lower than the current investor count. When a limit is met or exceeded existing investors are still able to receive tokens, but new investors are blocked.

`IssuingEntity.`**`setCountry`**(*uint16 _country*, *bool _allowed*, *uint8 _minRating*, *uint32[8] _limits*)

Approve or restrict a country, and/or modify it's minimum investor rating and investor limits.

- `_country`: The code of the country to modify
- `_allowed`: Permission bool
- `_minRating`: The minimum rating required for an investor in this country to hold tokens. Cannot be zero.

- `_limits`: A uint32[8] array of investor limits for this country.

`IssuingEntity.`**`setCountries`**(*uint16[] _country*, *bool _allowed*, *uint8[] _minRating*, *uint32[] _limit*)

Approve or restrict many countries at once.

- `_countries`: An array of country codes to modify
- `_allowed`: Permission bool
- `_minRating`: Array of minimum investor ratings for each country.
- `_limits`: Array of total investor limits for each country.

Each array must be the same length. The function will iterate through them at the same time: `_countries[0]` will require rating `_minRating[0]` and have a total investor limit of `_limits[0]`.

This method is useful when approving many countries that do not require specific limits based on investor ratings. When you require specific limits for each rating, use `IssuingEntity.setCountry`.

`IssuingEntity.`**`setInvestorLimits`**(*uint32[8] _limits*)

Sets total investor limits, irrespective of country.

`IssuingEntity.`**`getInvestorCounts`**()

Returns the sum total investor counts and limits for all countries and issuances related to this contract.

`IssuingEntity.`**`getCountry`**(*uint16 _country*)

Returns the minimum rating, investor counts and investor limits for a given country.

7.3.7 Document Verification

`IssuingEntity.`**`setDocumentHash`**(*string _documentID*, *bytes32 _hash*)

Creates an on-chain record of the hash of a legal document.

Once a hash is recorded, the issuer can distrubute the document electronically and investors can verify the authenticity by generating the hash themselves and comparing it to the blockchain record.

`IssuingEntity.`**`getDocumentHash`**(*string _documentID*)

Returns a recorded document hash.

7.3.8 Modules

The issuer may use these methods to attach or detach modules to this contract or any associated token contract.

See the *Modules* documentation for information module funtionality and development.

`IssuingEntity.`**`attachModule`**(*address _target*, *address _module*)

Attaches a module.

- `_target`: The address of the contract to associate the module to.
- `_module`: The address of the module contract.

`IssuingEntity.`**`detachModule`**(*address _target*, *address _module*)

Detaches a module. A module may call to detach itself, but not other modules.

`IssuingEntity.`**`isActiveModule`**(*address _module*)

Returns true if a module is currently active on the contract. Modules that are active on a token will return false.

7.4 KYCRegistrar

KYCRegistrar contracts are registries that hold information on the identity, region, and rating of investors.

Registries may be maintained by a single entity, or a federation of entities where each are approved to provide identification services for their specific jurisdiction. The contract owner can authorize other entities to add investors within specified countries.

Contract authorities associate addresses to ID hashes that denote the identity of the investor who owns the address. More than one address may be associated to the same hash. Anyone can call `KYCRegistrar.getID` to see which hash is associated to an address, and then using this ID call functions to query information about the investor's region and accreditation rating.

Registry contracts implement a variation of the standard *MultiSig Implementation* functionality used in other contracts within the protocol. This document assumes familiarity with the standard multi-sig implementation, and will only highlight the differences.

It may be useful to also view the KYCRegistrar.sol source code while reading this document.

7.4.1 Deployment

The **owner** is declared during deployment. The owner is the highest contract authority, impossible to restrict and the only entity capable of creating or restricting other authorities on the contract.

`KYCRegistrar.`**`constructor`**(*address[] _owners*, *uint32 _threshold*)

- `_owners`: One or more addresses to associate with the contract owner. The address deploying the contract is not implicitly included within the owner list.
- `_threshold`: The number of calls required for the owner to perform a multi-sig action. Cannot exceed the length of `_owners`.

The ID of the owner is generated as a keccak of the contract address and available from the public getter `ownerID`.

7.4.2 Working with Authorities

Authorities are known, trusted entities that are permitted to add, modify, or restrict investors within the registrar. Authorities are assigned a unique ID and associated with one or more addresses.

Only the owner may add, modify or restrict other authorities.

`KYCRegistrar.`**`addAuthority`**(*address[] _addr*, *uint16[] _countries*, *uint32 _threshold*)

Creates a new authority.

- `_owners`: One or more addresses to associate with the authority
- `_countries`: Countries that the authority is approved to act in
- `_threshold`: The number of calls required for the authority to perform a multi-sig action. Cannot exceed the length of `_owners`

Authorities do not require explicit permission to call any contract functions. However, they may only add, modify or restrict investors in countries that they have been approved to operate in.

Once an authority has been designated they may use `KYCRegistrar.registerAddresses` or `KYCRegistrar.restrictAddresses` to modify their associated addresses.

`KYCRegistrar.`**`setAuthorityCountries`**(*bytes32 _id*, *uint16[] _countries*, *bool _auth*)

Modifies the country permissions for an authority. Use `_auth` to determine if the call is permissive or restrictive.

`KYCRegistrar.`**`setAuthorityThreshold`**(*bytes32 _id*, *uint32 _threshold*)

Modifies the multisig threshold requirement for an authority. Can be called by any authority to modify their own threshold, or by the owner to modify the threshold for anyone.

You cannot set the threshold higher than the number of associated, unrestricted addresses for the authority.

`KYCRegistrar.`**`setAuthorityRestriction`**(*bytes32 _id*, *bool _permitted*)

Modifies the permitted state of an authority.

If an authority has been compromised or found to be acting in bad faith, the owner may apply a broad restriction upon them with this method. This will also restrict every investor that was approved by the authority.

A list of investors that were approved by the restricted authority can be obtained by looking at `NewInvestor` and `UpdatedInvestor` events. Once the KYC/AML of these investors has been re-verified, the restriction upon them may be removed by calling either `KYCRegistrar.updateInvestor` or `KYCRegistrar.setInvestorAuthority` to change which authority they are associated with.

7.4.3 Working with Investors

Investors are natural persons or legal entities who have passed KYC/AML checks and are approved to send and receive security tokens.

Each investor is assigned a unique ID and is associated with one or more addresses. They are also assigned an expiration time for their rating. This is useful in jurisdictions where accreditation status requires periodic reconfirmation.

Authorites may add, modify, or restrict investors in any country that they have been approved to operate in by the owner. See the *Investor Data Standards* documentation for detailed information on how this information is generated and formatted.

`KYCRegistrar.`**`generateID`**(*string _idString*)

Returns the keccak hash of the supplied string. Can be used by an authority to generate an investor ID hash from their KYC information.

`KYCRegistrar.`**`addInvestor`**(*bytes32 _id*, *uint16 _country*, *bytes3 _region*, *uint8 _rating*, *uint40 _expires*, *address[] _addr*)

Adds an investor to the registrar.

- `_id`: Investor's bytes32 ID hash
- `_country`: Investor country code
- `_region`: Investor region code
- `_rating`: Investor rating code
- `_expires`: The epoch time that the investor rating is valid until
- `_addr``: One or more addresses to associate with the investor

Similar to authorities, addresses associated with investors can be modified by calls to `KYCRegistrar.registerAddresses` or `KYCRegistrar.restrictAddresses`.

`KYCRegistrar.`**`updateInvestor`**(*bytes32 _id*, *bytes3 _region*, *uint8 _rating*, *uint40 _expires*)

Updates information on an existing investor.

Due to the way that the investor ID is generated, it is not possible to modify the country that an investor is associated with. An investor who changes their legal country of residence will have to resubmit KYC, be assigned a new ID, and transfer their tokens to a different address.

`KYCRegistrar.`**`setInvestorRestriction`**(*bytes32 _id*, *bool _permitted*)

Modifies the restricted status of an investor. An investor who is restricted will be unable to send or receive tokens.

`KYCRegistrar.`**`setInvestorAuthority`**`(`*bytes32[] _id*, *bytes32 _authID*`)`

Modifies the authority that is associated with one or more investors.

This method is only callable by the owner. It can be used after an authority is restricted, to remove the implied restriction upon investors that were added by that authority.

7.4.4 Adding and Restricting Addresses

Each authority and investor has one or more addresses associated to them. Once an address has been assigned to an ID, this association may never be removed. If an association were removed it would then be possible to assign that same address to a different investor. This could be used to circumvent transfer restrictions on tokens, allowing for non-compliant token ownership.

In situations of a lost or compromised private key the address may instead be flagged as restricted. In this case any tokens in the restricted address can be retrieved using another associated, unrestricted address.

`KYCRegistrar.`**`registerAddresses`**`(`*bytes32 _id*, *address[] _addr*`)`

Associates one or more addresses to an ID, or removes restrictions imposed upon already associated addresses.

If the ID belongs to an authority, this method may only be called by the owner. If the ID is an investor, it may be called by any authority permitted to work in that investor's country.

`KYCRegistrar.`**`restrictAddresses`**`(`*bytes32 _id*, *address[] _addr*`)`

Restricts one or more addresses associated with an ID.

If the ID belongs to an authority, this method may only be called by the owner. If the ID is an investor, it may be called by any authority permitted to work in that investor's country.

When restricing addresses associated to an authority, you cannot reduce the number of addresses such that the total remaining is lower than the multi-sig threshold value for that authority.

7.4.5 Getting Investor Info

There are a variey of getter methods available for issuers and custodians to query information about investors. In some cases these calls will revert if no investor data is found.

The following calls will not revert, instead returning `false` or an empty result:

`KYCRegistrar.`**`isRegistered`**`(`*bytes32 _id*`)`

Returns a boolean to indicate if an ID is known to the registrar contract. No permissioning checks are applied.

`KYCRegistrar.`**`getID`**`(`*address _addr*`)`

Given an address, returns the investor or authority ID associated to it. If there is no association it will return an empty bytes32.

`KYCRegistrar.`**`isPermitted`**`(`*address _addr*`)`

Given an address, returns a boolean to indicate if this address is permitted to transfer based on the following conditions:

- Is the registring authority restricted?
- Is the investor ID restricted?
- Is the address restricted?
- Has the investor's rating expired?

`KYCRegistrar.`**`isPermittedID`**`(`*address _addr*`)`

Returns a transfer permission boolean similar to `KYCRegistrar.isPermitted`, without a check on a specific address.

The remaining calls **will revert under some conditions**:

`KYCRegistrar.`**`getInvestor`**(*address _addr*)

Returns the investor ID, permission status (based on the input address), rating, and country code for an investor.

Reverts if the address is not registered.

Note: This function is designed to maximize gas efficiency when calling for information prior to performing a token transfer.

`KYCRegistrar.`**`getInvestorByID`**(*bytes32 _id*)

Returns the permission status, rating, and country code for an investor ID. Used by Custodians to check permission for an investor where there is no specific address associated to the action.

Reverts if the ID is not registered.

`KYCRegistrar.`**`getInvestors`**(*address _from*, *address _to*)

The two investor version of `KYCRegistrar.getInvestor`. Also used to maximize gas efficiency.

`KYCRegistrar.`**`getInvestorsByID`**(*bytes32 _fromID*, *bytes32 _toID*)

The two investor version of `KYCRegistrar.getInvestorByID`.

`KYCRegistrar.`**`getRating`**(*bytes32 _id*)

Returns the investor rating number for a given ID.

Reverts if the ID is not registered.

`KYCRegistrar.`**`getRegion`**(*bytes32 _id*)

Returns the investor region code for a given ID.

Reverts if the ID is not registered.

`KYCRegistrar.`**`getCountry`**(*bytes32 _id*)

Returns the investor country code for a given ID.

Reverts if the ID is not registered.

`KYCRegistrar.`**`getExpires`**(*bytes32 _id*)

Returns the investor rating expiration date (in epoch time) for a given ID.

Reverts if the ID is not registered or the rating has expired.

7.5 Custodian

Custodian contracts allow approved entities to hold tokens on behalf of multiple investors. Each custodian must be individually approved by an issuer before they can receive tokens.

There are two broad categories of custodians:

- **Owned** custodians are contracts that are controlled and maintained by a known legal entity. Examples of owned custodians include broker/dealers or centralized exchanges.
- **Autonomous** custodians are contracts without an owner. Once deployed there is no authority capable of exercising control over the contract. Examples of autonomous custodians include escrow services, privacy protocols and decentralized exchanges.

It may be useful to view source code for the following contracts while reading this document:

- IMiniCustodian.sol: The minimum contract interface required for a Custodian contract to interact with an IssuingEntity contract.

- Owned.sol: Standard owned custodian contract with Multisig and Modular functionality.
- Escrow.sol: An example autonomous custodian implementation, providing on-chain enforceable escrow.

Warning: An issuer should not approve a Custodian contract if it's source code cannot be verified, or it is using a non-standard implementation that has not undergone a thorough audit. Inaccurate balance reporting could enable a range of exploits. The SFT protocol includes a standard owned Custodian contract that allows for modular customization without introducing security concerns.

7.5.1 How Custodians Work

Custody and Beneficial Ownership

Custodians interact with an issuer's investor counts differently from regular investors. When an investor transfers a balance into a custodian it does not increase the overall investor count, instead the investor is now included in the list of beneficial owners represented by the custodian. Even if the investor now has a balance of 0 in their own wallet, they will be still be included in the issuer's investor count.

Custodian transfer functions include a boolean `_stillOwner`. When set to true, even if an investor's balance is at 0 as a result of a transfer, that investor will still be included in the list of beneficial owners within the custodian. This allows the custodian to continue to reserver the slot within the investor count, which is useful in a situation such as a secondary market where an investor may be moving in and out of the position many times over a short period.

The value of `_stillOwner` is only checked when a transfer results in a 0 balance for the sender. If set to false during a transfer where the investor's final balance is greater than zero, the transfer will succeed but the beneficial owner status will not be released.

Token Transfers

There are three types of token transfers related to Custodians.

- **Inbound**: transfers from an investor into the Custodian contract.
- **Outbound**: transfers from the Custodian contract to an investor's wallet.
- **Internal**: transfers involving a change of beneficial ownership records within the Custodian contract. This is the only type of transfer that involves a change of ownership of the token.

In order to perform these transfers, Custodian contracts interact with IssuingEntity and SecurityToken contracts via the following methods. None of these methods are user-facing; if you are only using the standard Custodian contracts within the protocol you can skip the rest of this section.

Inbound

Inbound transfers are those where an investor sends tokens into the Custodian contract. They are initiated in the same way as any other transfer, by calling the `SecurityToken.transfer` or `SecurityToken.transferFrom` methods. Inbound transfers do not register a change of beneficial ownership, however if the sender previously had a 0 balance with the custodian they will be added to that custodian's list of beneficial owners.

During an inbound transfer the following method is be called in the custodian contract:

`IMiniCustodian.`**`receiveTransfer`**(*address _token*, *bytes32 _id*, *uint256 _value*)

- `_token`: Token addresss being transferred to the the Custodian.

- `_id`: Sender ID.
- `_value`: Amount being transferred.

Called from `IssuingEntity.transferTokens`. Used to update the custodian's balance and investor counts. Revert or return `false` to block the transfer.

Outbound

Outbound transfers are those where tokens are sent from the Custodian contract to an investor's wallet. Depending on the type of custodian and intended use case they may be initiated in several different ways.

Internally, the Custodian contract sends tokens back to an investor using the normal `SecurityToken.transfer` method. No change of beneficial ownership is recorded.

The IssuingEntity contract does not keep a specific record of investor balances within each Custodian. If a transfer removes an investor from the Custodian's list of beneficial owners, it should be followed by a call to `IssuingEntity.releaseOwnership`. See the *IssuingEntity* documentation for information about this method.

Internal

Internal transfers involve a change of beneficial ownership records within the Custodian contract. Tokens do not enter or leave the Custodian contract, but a call is made to the corresponding token contract to verify that the transfer is permitted.

The Custodian contract can call the following SecurityToken methods to register internal transfers.

`SecurityToken.`**`checkTransferCustodian`**`(`*bytes32[2] _id*, *bool _stillOwner*`)`

Returns true if the Custodian is permitted to perform an internal transfer of ownership for this token.

- `_id`: Array of sender and recipient IDs.
- `_stillOwner`: Is the sender still a beneficial owner?

`SecurityToken.`**`transferCustodian`**`(`*bytes32[2] _id*, *uint256 _value*, *bool _stillOwner*`)`

Modifies investor counts and ownership records based on an internal transfer of ownership within the Custodian contract.

- `_id`: Array of sender and recipient IDs.
- `_value`: Amount of tokens being transferred
- `_stillOwner`: Is the sender still a beneficial owner?

Minimal Implementation

The `IMiniCustodian` interface defines a minimal implementation required for custodian contracts to interact with an IssuingEntity contract. Notably absent from this interface is a way for tokens to transfer out of the contract. Depending on the type of custodian and intended use case, outgoing transfers may be implemented in different ways.

`IMiniCustodian.`**`ownerID`**`()`

Public bytes32 hash representing the owner of the contract.

`IMiniCustodian.`**`balanceOf`**`(`*address _token*, *bytes32 _id*`)`

View function to query the balance of an investor for a specific token.

- `_token`: SecurityToken address
- `_id`: Investor ID

While there is no strict requirement for a Custodian to maintain an on-chain record of investor balances, this information is necessary if the custodian is to e.g. allow investors to claim dividends or exercise voting rights based on held balances. As such, balances should always be accurately recorded on-chain unless there is a use case that requires otherwise.

`IMiniCustodian.`**`isBeneficialOwner`**`(`*address _issuer*, *bytes32 _id*`)`

Checks if an investor is on the custodian's list of beneficial owners for this issuer.

- `_issuer`: IssuingEntity contract address
- `_id`: Investor ID

`IMiniCustodian.`**`receiveTransfer`**`(`*address _token*, *bytes32 _id*, *uint256 _value*`)`

- `_token`: Token addresss being transferred to the the Custodian.
- `_id`: Sender ID.
- `_value`: Amount being transferred.

Called from `IssuingEntity.transferTokens` when tokens are being sent into the Custodian contract. It should be used to update the custodian's balance and investor counts. Revert or return `false` to block the transfer.

7.5.2 Owned Custodians

Owned custodians are contracts that are controlled and maintained by a known legal entity. Examples of owned custodians include broker/dealers or centralized exchanges.

Owned Custodian contracts include the standard SFT protocol *MultiSig Implementation* and *Modules* functionality. See the respective documents for detailed information on these components.

Deployment

The constructor declares the owner as per standard *MultiSig Implementation*.

`OwnedCustodian.`**`constructor`**`(`*address[] _owners*, *uint32 _threshold*`)`

- `_owners`: One or more addresses to associate with the contract owner. The address deploying the contract is not implicitly included within the owner list.
- `_threshold`: The number of calls required for the owner to perform a multi-sig action.

The ID of the owner is generated as a keccak of the contract address and available from the public getter `ownerID`.

Token Transfers

Investor balances for each token are tracked on-chain. Investors may send tokens into the contract, but only the contract owner has the authority to initiate internal and outbound transfers.

To maintain accurate beneficial owner records, custodians must initiate all token transfers through the contract instead of calling `SecurityToken.transfer` directly.

`OwnedCustodian.`**`checkTransferInternal`**`(`*address _token*, *bytes32 _fromID*, *bytes32 _toID*, *uint256 _value*, *bool _stillOwner*`)`

Checks if an internal transfer is permitted.

- `_token`: SecurityToken address

- `_fromID`: Sender ID
- `_toID`: Receiver ID
- `_value`: Amount to transfer
- `_stillOwner`: Is the sender still a beneficial owner for this issuer?

`OwnedCustodian.`**`transferInternal`**(*address _token*, *bytes32 _fromID*, *bytes32 _toID*, *uint256 _value*, *bool _stillOwner*)

- `_token`: SecurityToken address
- `_fromID`: Sender ID
- `_toID`: Receiver ID
- `_value`: Amount to transfer
- `_stillOwner`: Is the sender still a beneficial owner for this issuer?

`OwnedCustodian.`**`transfer`**(*address _token*, *address _to*, *uint256 _value*, *bool _stillOwner*)

Transfers tokens out of the Custodian contract.

- `_token`: SecurityToken address
- `_to`: Investor address to send tokens to
- `_value`: Amount to transfer
- `_stillOwner`: Is the receiver still a beneficial owner for this issuer?

Modules

See the *Modules* documentation for information module funtionality and development.

Note: For Custodians that require bespoke functionality it is preferrable to attach modules than to modify the core contract. Inaccurate balance reporting could enable a range of exploits, and so Issuers should be very wary of permitting any Custodian that uses a non-standard contract.

`OwnedCustodian.`**`attachModule`**(*address _module*)

Attaches a module to the custodian.

`OwnedCustodian.`**`detachModule`**(*address _module*)

Detaches a module. A module may call to detach itself, but not other modules.

`OwnedCustodian.`**`isActiveModule`**(*address _module*)

Returns true if a module is currently active on the contract.

7.5.3 Autonomous Custodians

Autonomous custodians have no owner. Once deployed there is no authority capable of exercising control over the contract. Examples of autonomous custodians include escrow services, privacy protocols and decentralized exchanges.

Unlike the owned Custodian there is no single common approach for an autonomous custodian. Their use cases vary significantly such that we cannot effectively define a standard interface.

At present SFT contains one autonomous Custodian, an on-chain escrow contract meant to serve as a proof of concept. We intend to develop and audit additional autonomous Custodian contracts to expand the range of functionality in the protocol.

7.6 Security Considerations

A fully realized implementation of the SFT protocol involves many interconnected contracts maintained by different entities. As such, it is important to anticipate and know how to deal with events where another party is compromised or found to be acting in bad faith.

We have compiled a list of possible scenarios and solutions below. If you can imagine an issue that is not mentioned here, please contact us so we can discuss it and add it to the list.

Note: In this section we provide technical solutions to problems, however many of these situations will also have a legal component. The nature of security tokens means that every involved entity is easily identified, so when someone is acting in bad faith it is possible that resolution will be the result of a court order. Issuers must keep in mind that although technically they can transfer any investor's tokens without approval, this does not mean that legally they are always allowed to.

7.6.1 Investor Changes Country

An investor who changes their legal country of residence will necessarily alter their ID hash. In this case the investor should resubmit their KYC/AML to an registrar active within the new country, receive a new ID hash attached to a new address, and transfer their tokens from their old address to the new one. Their old ID may then be restricted.

7.6.2 Investor is Sanctioned

If an investor is sanctioned or otherwise has their assets legally frozen, a registrar can use `KYCRegistrar.setInvestorRestriction` to block them from transferring any of their tokens.

7.6.3 Court Ordered Transfer of Assets

In cases such as a lawsuit or the execution of a will, an issuer may be legally required to perform a token transfer. This is possible using `SecurityToken.transferFrom`.

7.6.4 Lost Investor Private Key

An investor who has lost a private key should contact the registry authority and verify their identity off-chain. The authority can restrict the address of the lost key with `KYCRegistrar.restrictAddresses`, then add one or more new addresses with `KYCRegistrar.registerAddresses`. The investor may retrieve tokens from the lost address either with assistance from the issuer, or by using the `SecurityToken.transferFrom` method.

7.6.5 Compromised Investor Private Key

If an investor's private key is hacked, they should contract the registrar immediately to have the hacked address restricted. If tokens were transferred from the restricted address before it was blocked, the response will depend on the nature of the transfers:

- If tokens were sent directly to another investor, the issuer can use `IssuingEntity.setInvestorRestriction` to restrict the recipient until a legal resolution is reached. They can then use `SecurityToken.transferFrom` to return the tokens to the original address.

- If tokens were sent directly into a centralized exchange, the exchange must be notified immediately. Whether the exchange can help will depend on if the tokens were sold or not, and if yes, whether the funds from the sale were withdrawn and where they were sent.

7.6.6 Compromised Registrar Authority

If a registrar authority has been hacked or found to be acting in bad faith, the owner of the registrar may apply a broad restriction upon them using `setAuthorityRestriction`. This will also restrict every investor that was approved by this authority.

A list of investors that were approved by the restricted authority can be obtained from `NewInvestor` and `UpdatedInvestor` events. Once the KYC/AML of these investors has been re-verified, the restriction upon them may be removed by calling either `KYCRegistrar.updateInvestor` or `KYCRegistrar.setInvestorAuthority`.

7.6.7 Compromised Registrar

In a case where a registrar contract is so thoroughly compromised that an issuer deems it can no longer be trusted, the issuer can remove the registrar by calling `IssuingEntity.setRegistrar`. This will also restrict every investor that was approved by this registry. These investors will have to KYC via a different registrar in order to be able to transfer their tokens.

7.6.8 Compromised Issuer/Custodian Authority

If an IssungEntity or Custodian authority is hacked or found to be acting in bad faith, the owner can restrict the authority using `IssuingEntity.setAuthorityApprovedUntil`. Further actions will depend on the severity of the actions performed by the compromised account prior to it being frozen.

This situation can be mitigated against with multi-sig requirements, method permissioning, and temporary approval to authorities.

7.6.9 Compromised Custodian

If a custodian is hacked or found to be acting in bad faith, an issuer may block them with `IssuingEntity.setInvestorRestriction`. They may then use `IssuingEntity.setBeneficialOwners` to remove the custodian from the list of beneficial owners, and `SecurityToken.transferFrom` to seize any tokens held by the custodian.

A list of beneficial owners can be obtained by filtering for the `BeneficialOwnerSet` event.

7.6.10 Compromised Issuer

As the issuer is the highest authority over their own tokens, a fully compromised issuer presents a challenging situation to overcome. Issuers should always follow strict security practices including keeping the original owner private keys in cold storage, isolating function authority via permissioning, and using strict multi-sig requirements.

If an IssuingEntity contract is compromised the best course of action will be to immediately notify all investors and custodians and halt secondary trading. The issuer will have to deploy new contracts and reissue tokens based on a determined historic state of the blockchain.

7.7 MultiSig Implementation

IssuingEntity and *Custodian* contracts both implement a common multisig functionality that allows the contract owner to designate other authorities the ability to call specific admin-level contract methods.

KYCRegistrar contracts use a slightly modified implementation.

It may be useful to also view the MultiSig.sol source code while reading this document.

7.7.1 Deployment

The **owner** is declared during deployment. The owner is the highest contract authority, impossible to restrict and the only entity capable of creating or restricting other authorities on the contract.

`MultiSig.`**`constructor`**(*address[] _owners*, *uint32 _threshold*)

- `_owners`: One or more addresses to associate with the contract owner. The address deploying the contract is not implicitly included within the owner list.
- `_threshold`: The number of calls required for the owner to perform a multi-sig action.

The ID of the owner is generated as a keccak of the contract address and available from the public getter `ownerID`.

The owner has the highest level of control over the contract. Associated addresses may always call any admin-level functionality.

7.7.2 Working With Authorities

Authorities are entities that are permitted to call admin-level methods within a contract. They are assigned a unique ID that is associated with one or more addresses.

Authorities differ from the owner in that they must be explicitly approved to call functions within the contract. These permissions may be modified by the owner via a call to `MultiSig.setAuthoritySignatures`. You can check if an authority is permitted to call a specific function with the view function `MultiSig.isApprovedAuthority`.

Only the owner may add, modify or restrict other authorities.

`MultiSig.`**`addAuthority`**(*address[] _addr*, *bytes4[] _signatures*, *uint32 _approvedUntil*, *uint32 _threshold*)

Approves a new authority.

- `_addr`: One or more addresses to associated with the authority.
- `_signatures`: Function signatures that this authority is permitted to call.
- `_approvedUntil`: The epoch time that this authority is permitted to make calls until. To approve an authority forever, set it to the highest possible uint32 value of 4294967296 (February, 2106).
- `_threshold`: The number of calls required by this authority to perform a multi-sig action.

`MultiSig.`**`setAuthorityApprovedUntil`**(*bytes32 _id*, *uint32 _approvedUntil*)

Modifies the date an authority is approved to act until.

The owner can restrict an authority by calling this function and setting `_approvedUntil` to 0.

`MultiSig.`**`setAuthoritySignatures`**(*bytes32 _id*, *bytes4[] _signatures*, *bool _allowed*)

Modifies call permissions for an authority.

`MultiSig.`**`setAuthorityThreshold`**`(`*bytes32 _id*, *uint32 _threshold*`)`

Modifies the multisig threshold requirement for an authority. Can be called by any authority to modify their own threshold, or by the owner to modify the threshold for anyone.

`MultiSig.`**`addAuthorityAddresses`**`(`*bytes32 _id*, *address[] _addr*`)`

Associates addresses with an authority. Can be called by any authority to add to their own addresses, or by the owner to add addresses for any authority. Can also be used to re-approve a previously restricted address that is already associated to the authority.

`MultiSig.`**`removeAuthorityAddresses`**`(`*bytes32 _id*, *address[] _addr*`)`

Restricts addresses that are associated with an authority. Can be called by any authority to restrict to their own addresses, or by the owner to restrict addresses for any authority.

Once an address has been assigned to an authority, this association may never be removed. If an association were removed it would then be possible to assign that same address to a different investor. This could be used to circumvent various contract restricions.

`MultiSig.`**`isApprovedAuthority`**`(`*address _addr*, *bytes4 _sig*`)`

Returns true if the authority associated with the given address is permitted to call the method with the given signature.

7.7.3 Implementing MultiSig

Multisig functionality can be implemented within any contract method as well as in external contracts.

`MultiSig.`**`_checkMultiSig`**`()`

Internal function, used to implement multisig within a function in the same contract.

All multi-sig functions return a single boolean to indicate if the threshold was met and the call succeeded. Functions that implement multi-sig include the following line of code, either at the start orafter the initial require statements:

```
if (!_checkMultiSig()) return false;
```

Calls that fail to meet the threshold will trigger an event `MultiSigCall` which includes the current call count and the threshold value. Once a caller meets the threshold the event `MultiSigCallApproved` will trigger, the call will execute, and the call count will be reset to zero.

The number of calls to a function is recorded using a keccak hash of the call data. As such, it is required that each calling address format their call data in exactly the same way.

Repeating a multi-sig call from the same address before reaching the threshold will revert.

`MultiSig.`**`checkMultiSigExternal`**`(`*bytes4 _sig*, *bytes32 _callHash*`)`

External function, used to implement multisig in an different contract.

- `_sig`: The original function signature being called
- `_callHash`: a keccak hash of the original calldata

Use the following code to implement this in an external contract:

```
bytes32 _callHash = keccak256(msg.data);
if (!MultiSigContract.checkMultiSigExternal(msg.sig, _callHash)) {
    return false;
}
```

This function relies on `tx.origin` to verify that the original caller is an approved authority. Permissions are checked against the signature value in the same way as with an internal call. The recorded keccak hash of the call is formed by joining the address of the calling contract, the signature, and the supplied call hash. As such it is impossible to exploit the external call to advance the count on internal multisig events.

Warning: If an external contract includes a function with the same signature as one inside the multisig contract, it will be impossible to set unique permissions for each function. Developers and auditors of external contracts should always keep this in mind.

7.8 Modules

Modules are contracts that hook into various methods in *IssuingEntity*, *SecurityToken* and *Custodian* contracts. They may be used to add custom permissioning logic or extra functionality.

It may be useful to view source code for the following contracts while reading this document:

- Modular.sol: Inherited by modular contracts. Provides functionality around attaching, detaching, and calling modules.
- ModuleBase.sol: Inherited by modules. Provide required functionality for modules to be able to attach or detach.
- IModules.sol: Interfaces outlining standard module functionality. Includes inputs for all possible hook methods.

Note: In order to minimize gas costs, modules should be attached only when their functionality is required and detached as soon as they are no longer needed.

Warning: Depending on the hook and permission settings, modules may be capable of actions such as blocking transfers, moving investor tokens and altering the total supply. Only attach a module that has been properly audited, ensure you understand exactly what it does, and be **very** wary of any module that requires permissions outside of it's documented behaviour.

7.8.1 Attaching and Detaching

Modules are attached or detached via methods `attachModule` and `detachModule` in the inheriting contracts. See the *IssuingEntity* and *Custodian* documentation implementations.

Token modules are attached and detached via the associated IssuingEntity contract.

All contracts implementing modular functionality will also include the following method:

`Modular.`**`isActiveModule`**`(`*address _module*`)`

Returns true if a module is currently active on the contract.

Modules that are attached to an IssuingEntity are also considered active on any tokens belonging to that issuer.

Modules include the following getters:

`ModuleBase.`**`getOwner`**`()`

Returns the address of the parent contract that the module has been attached to.

`ModuleBase.`**`name`**`()`

Returns a string name of the module.

7.8.2 Permissioning and Functionality

Modules introduce functionality in two ways:

- **Hooks** are points within the parent contract's methods where the module will be called. They can be used to introduce extra permissioning requirements or record additional data.
- **Permissions** are methods within the parent contract that the module is able to call into. This can allow actions such as adjusting investor limits, transferring tokens, or changing the total supply.

In short: hooks involve calls from a parent contract into a module, permissions involve calls from a module into the parent contract.

Hooks and permissions are set the first time a module is attached by calling the following method:

`ModuleBase.`**`getPermissions`**`()`

Returns two `bytes4[]`:

- `hooks`: Array of method signatures within the module that the parent will call to.
- `permissions`: Array of method signatures within the parent contract that the module is permitted to call.

Before attaching a module, be sure to check the return value of this function and compare the requested hook points and permissions to those that would be required for the documented functionality of the module. For example, a module intended to block token transfers should not require permission to mint new tokens.

7.8.3 Hooking into Methods

The available hook points varies depending on the type of parent contract.

SecurityToken

`STModule.`**`checkTransfer`**`(`*address[2] _addr*, *bytes32 _authID*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*`)`

- Hook signature: `0x70aaf928`

Called by `SecurityToken.checkTransfer` to verify if a transfer is permitted.

- `_addr`: Sender and receiver addresses.
- `_authID`: ID of the authority who wishes to perform the transfer. It may differ from the sender ID if the check is being performed prior to a `transferFrom` call.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver countriy codes.
- `_value`: Amount to be transferred.

`STModule.`**`transferTokens`**`(`*address[2] _addr*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*`)`

- Hook signature: `0x35a341da`

Called after a token transfer has completed successfully with `SecurityToken.transfer` or `SecurityToken.transferFrom`.

- `_addr`: Sender and receiver addresses.

- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver country codes.
- `_value`: Amount that was transferred.

`STModule.`**`transferTokensCustodian`**(*address _custodian*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*)

- Hook signature: `0x6eaf832c`

Called after an internal custodian token transfer has completed with `Custodian.transferInternal`.

- `_custodian`: Address of the custodian contract.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver country codes.
- `_value`: Amount that was transferred.

`STModule.`**`balanceChanged`**(*address _addr*, *bytes32 _id*, *uint8 _rating*, *uint16 _country*, *uint256 _old*, *uint256 _new*)

- Hook signature: `0x4268353d`

Called after a balance has been directly modified by `SecurityToken.modifyBalance`. Calls to this method also modify the total supply.

- `_addr`: Address where balance has changed.
- `_id`: ID that the address is associated to.
- `_rating`: Investor rating.
- `_country`: Investor country code.
- `_old`: Previous token balance at the address.
- `_new`: New token balance at the address.

IssuingEntity

`IssuerModule.`**`checkTransfer`**(*address _token*, *bytes32 _authID*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*)

- Hook signature: `0x47fca5df`

Called by `IssuingEntity.checkTransfer` to verify if a transfer is permitted.

- `_token`: Address of the token to be transferred.
- `_authID`: ID of the authority who wishes to perform the transfer. It may differ from the sender ID if the check is being performed prior to a `transferFrom` call.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver countriy codes.
- `_value`: Amount to be transferred.

`IssuerModule.`**`transferTokens`**(*address _token*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*)

- Hook signature: `0x0cfb54c9`

Called after a token transfer has completed successfully with `SecurityToken.transfer` or `SecurityToken.transferFrom`.

- `_token`: Address of the token that was transferred.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver country codes.
- `_value`: Amount that was transferred.

`IssuerModule.`**`transferTokensCustodian`**(*address _token*, *address _custodian*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint256 _value*)

- Hook signature: `0x3b59c439`

Called after an internal custodian token transfer has completed with `Custodian.transferInternal`.

- `_token`: Address of the token that was transferred.
- `_custodian`: Address of the custodian contract.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver country codes.
- `_value`: Amount that was transferred.

`IssuerModule.`**`balanceChanged`**(*address _token*, *bytes32 _id*, *uint8 _rating*, *uint16 _country*, *uint256 _old*, *uint256 _new*)

- Hook signature: `0x4268353d`

Called after a balance has been directly modified by `SecurityToken.modifyBalance`. Calls to this method also modify the total supply.

- `_token`: Token address where balance has changed.
- `_id`: ID of the investor who's balance changed.
- `_rating`: Investor rating.
- `_country`: Investor country code.
- `_old`: Previous investor balance (across all tokens).
- `_new`: New investor balance (across all tokens).

Custodian

`CustodianModule.`**`sentTokens`**(*address _token*, *bytes32 _id*, *uint256 _value*, *bool _stillOwner*)

- Hook signature: `0x31b45d35`

Called after tokens have been transferred out of a Custodian via `Custodian.transfer`.

- `_token`: Address of token that was sent.
- `_id`: ID of the recipient.
- `_value`: Number of tokens that were sent.
- `_stillOwner`: Is the recipient still a beneficial owner for this token?

`CustodianModule.`**`receivedTokens`**(*address _token*, *bytes32 _id*, *uint256 _value*, *bool _newOwner*)

- Hook signature: `0xa0e7f751`

Called after a tokens have been transferred into a Custodian.

- `_token`: Address of token that was received.
- `_id`: ID of the sender.
- `_value`: Number of tokens that were received.

`CustodianModule.`**`internalTransfer`**(*address _token*, *bytes32 _fromID*, *bytes32 _toID*, *uint256 _value*, *bool _stillOwner*)

- Hook signature: `0x7054b724`

Called after an internal transfer of ownership within the Custodian contract via `Custodian.transferInternal`.

- `_token`: Address of token that was received.
- `_fromID`: ID of the sender.
- `_toID`: ID of the recipient.
- `_value`: Number of tokens that were received.
- `_stillOwner`: Is the sender still a beneficial owner for this token?

`CustodianModule.`**`ownershipReleased`**(*address _issuer*, *bytes32 _id*)

- Hook signature: `0x054d1c76`

Called after an investor's beneficial ownership status has been released within the Custodian contract via `Custodian.releaseOwnership`.

- `_issuer`: IssuingEntity contract address
- `_id`: Investor ID

7.8.4 Calling Parent Methods

Once attached, modules may call into methods in the parent contract where they have been given permission.

Note: When a module calls into the parent contract, it will still trigger any of it's own methods hooked into the called method. With poor contract design you can create infinite loops and effectively break the parent contract functionality as long as the module remains attached.

SecurityToken

Any module applied to an IssuingEntity contract may also be permitted to call methods on any token belonging to the issuer. See *SecurityToken* for more detailed information on these methods.

`SecurityToken.`**`transferFrom`**(*address _from*, *address _to*, *uint256 _value*)

- Permission signature: `0x23b872dd`

Transfers tokens between two addresses. A module calling `transferFrom` has the same level of authority as if the call was from the issuer.

Calling this method will also call any hooked in `checkTransfer` and `transferTokens` methods.

`SecurityToken.`**`modifyBalance`**`(`*address _owner*, *uint256 _value*`)`

- Permission signature: `0x250dea06`

Sets the balance of `_owner` to `_value` and modifies `totalSupply` accordingly. This method is only callable by a module.

Calling this method will also call any hooked in `balanceChanged` methods.

`SecurityToken.`**`detachModule`**`(`*address _module*`)`

- Permission signature: `0xbb2a8522`

Detaches a module. This method can only be called directly by a permitted module, for the issuer to detach a SecurityToken level module the call must be made via the IssuingEntity contract.

IssuingEntity

`IssuingEntity.`**`detachModule`**`(`*address _target*, *address _module*`)`

- Permission signature: `0xbb2a8522`

Detaches module contract `_module` from parent contract `_target`.

Custodian

See *Custodian* for more detailed information on these methods.

`Custodian.`**`transfer`**`(`*address _token*, *address _to*, *uint256 _value*, *bool _stillOwner*`)`

- Permission signature: `0x75219e4e`

Transfers tokens from the custodian to an investor.

Calling this method will also call any hooked in `sentTokens` methods.

`Custodian.`**`transferInternal`**`(`*address _token*, *bytes32 _fromID*, *bytes32 _toID*, *uint256 _value*, *bool _stillOwner*`)`

- Permission signature: `0x2965c868`

Transfers the ownership of tokens between investors within the Custodian contract.

Calling this method will also call any hooked in `internalTransfer` methods.

`Custodian.`**`releaseOwnership`**`(`*address _issuer*, *bytes32 _id*`)`

- Permission signature: `0xc07f6f8e`

Removes an investor from the Custodian's list of beneficial owners.

Calling this method will also call any hooked in `ownershipReleased` methods.

`Custodian.`**`detachModule`**`(`*address _module*`)`

- Permission signature: `0xbb2a8522`

Detaches a module.

7.8.5 Use Cases

The wide range of functionality that modules can hook into and access allows for many different applications. Some examples include: crowdsales, country/time based token locks, right of first refusal enforcement, voting rights, dividend payments, tender offers, and bond redemption.

We have included some sample modules on GitHub as examples to help understand module development and demonstrate the range of available functionality.

7.9 Investor Data Standards

The following generation and format standards should be followed across the SFT protocol to ensure interoperability between network participants.

7.9.1 Investor IDs

Investor IDs are stored as a bytes32 keccak256 hash of the investor's personally identifiable information.

For legal entities, the hash is generated from their Global Legal Entity Identifier (LEI):

> *The International Organization for Standardization (ISO) 1744 standard defines a set of attributes or legal entity reference data that are the most essential elements of identification. The Legal Entity Identifier (LEI) code itself is neutral, with no embedded intelligence or country codes that could create unnecessary complexity for users.*

For natural persons, a hash is produced from a concatenation of the following:

- Full legal name in all capital letters without spaces
- Date of Birth as DDMMYYYY
- Unique tax ID from current jurisdiction of residence

If any of the malleable fields are changed (via a legal name change or a change of home jurisdictions), the investor will be required to pass KYC/AML again and a new investor ID will be generated. Once KYC is passed, the tokens held in previous addresses must be transferred to addresses associated to the new investor ID. It is impossible to remove or change the ID association of an address.

7.9.2 Country Codes

Based on the ISO-3166-1 numeric standard. Country codes are stored as a uint16 and follow the standard exactly.

A CSV of country and region codes is available here.

7.9.3 Region Codes

Based on the ISO 3166-2 standard. Region codes are stored as a bytes3 and are generated in the following way:

1. Convert each character of the ISO 3166-2 code to it's hexadecimal ASCII code point
2. Concatenate the hex values
3. Pad right where necessary

A quick example to generate region codes using python:

```
iso3166 = "US-AL"[3:]
iso3166 = [hex(ord(i)).replace('0x','') for i in iso3166]
print("0x"+"".join(iso3166)).ljust(6, '0'))
```

- Original code: US-AL
- Resulting bytes32: 0x414c00

A CSV of country and region codes is available here.

7.10 Glossary

- **Authority**: A collection of one or more addresses permitted to call specific admin-level functionality in a multisig contract.
- **Custodian**: An entity that is approved to hold tokens on behalf of multiple investors. Common examples of custodians include broker/dealers and secondary markets.
- **Entity**: A participant in the SFT protocol. Entity may refer to natural persons or corporations.
- **Hook**: The point at which a module attaches to a method in a parent contract.
- **Issuer**: An entity that creates and sells security tokens.
- **Investor**: An entity that has passed KYC/AML checks and is able to hold and transfer security tokens.
- **Module**: A non-essential smart contract associated with an IssuingEntity or SecurityToken contract by an issuer, used to add extra transfer permissioning or handle on-chain governance events.
- **Owner**: The highest authority of a contract, set durin deployment. Only the owner is capable of creating or restricting other authorities on that contract.
- **Rating**: A number assigned to each investor that corresponds to their accreditation status.
- **Region**: Refers to the state, province, or other principal subdivision that an investor resides in.
- **Security Token**: An ERC-20 compliant token, created by an issuer, who's transferrability is restricted through on-chain logic.
- **Threshold**: The number of required calls from an authority to an admin-level function before it executes. This value cannot be greater the number of addresses associated with the authority.
- **Registrar**: A whitelist contract that associates ethereum addresses to specific investors.

Index