SFT Protocol Documentation

Benjamin Hauser, Alex Firmani

Sep 23, 2019

Contents

The Secured Financial Transaction Protocol (SFT) is a set of smart contracts, written in Solidity for the Ethereum blockchain, that allow for the tokenization of financial securities. It provides a robust, modular framework that is configurable for a wide range of jurisdictions, with consideration for real world needs based on today's existing markets. SFT favors handling as much permissioning logic on-chain as possible, in order to maximize transparency for all parties involved.

The SFT Protocol was developed by Ben Hauser of ZeroLaw Tech.

Note: Code starting with `$` is meant to be run in your terminal. Code starting with `>>>` is meant to run inside the Brownie console.

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:

- **Issuers** are entities that create and sell security tokens to fund their business operations.
- **Investors** are entities that have passed KYC/AML checks and are are able to hold or transfer security tokens.
- **Registrars** are trusted entities that provide KYC/AML services for network participants.
- **Custodians** hold tokens on behalf of investors without taking direct ownership. They may provide services such as escrow or custody, or 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 permissioning checks. 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 meet the needs of each individual issuer.

CHAPTER 2

Components

The SFT protocol is comprised of four core components:

1. *Token*
    - ERC20 compliant token contracts
    - Intended to represent a corporate shareholder registry in book entry or certificated form
    - Permissioning logic to enforce enforce legal and contractural restrictions around token transfers
    - Modular design allows for optional added functionality
2. *IssuingEntity*
    - Common owner contract for multiples classes of tokens created by the same issuer
    - Detailed on-chain cap table with granular permissioning capabilities
    - Modular design allows for optional added functionality
    - Multi-sig, multi-authority design provides increased security and permissioned contract management
3. *KYC*
    - Whitelists that provide identity, region, and accreditation information of investors based on off-chain KYC/AML verification
    - May be maintained by a single entity for a single token issuance, or a federation across multiple jurisdictions providing identity data for many issuers
    - Multi-sig, multi-authority design provides increased security and permissioned contract management
4. *Custodians*
    - Contracts that represent an entity approved to hold tokens on behalf of multiple investors
    - Deep integration with IssuingEntity to provide accurate on-chain investor counts
    - Multiple implementations allow for a wide range of functionality including escrow services, custody, and secondary trading of tokens
    - Modular design allows for optional added functionality

- Multi-sig, multi-authority design provides increased security and permissioned contract management

CHAPTER 3

Source Code

Many core components of the SFT Protocol are open sourced. You can view the code on GitHub.

CHAPTER 4

Testing and Deployment

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

To run the tests:

```
$ pytest test
```

CHAPTER 5

License

This project is licensed under the Apache 2.0 license.

CHAPTER 6

Contents

Keyword Index, *Glossary*

6.1 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 the brownie console:

```
$ brownie console
Brownie v1.0.0 - Python development framework for Ethereum

Brownie environment is ready.
>>> run('deployment')
```

This runs the `main` function in scripts/deployment.py which:

- Deploys `KYCRegistrar` from `accounts[0]`
- Deploys `IssuingEntity` from `accounts[0]`
- Deploys `SecurityToken` from `accounts[0]` with an initial authorized supply of 1,000,000 tokens
- Associates the contracts
- Approves `accounts[1:7]` 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 mint and transfer tokens:

```
>>> token = SecurityToken[0]
>>> token.mint(accounts[1], 1000, {'from': accounts[0]})

Transaction sent: 0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e
SecurityToken.mint confirmed - block: 13   gas used: 229092 (2.86%)
```

(continues on next page)

(continued from previous page)

```
<Transaction object
↪'0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e'>
>>>
>>> token.transfer(accounts[2], 1000, {'from': accounts[1]})

Transaction sent: 0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f
SecurityToken.transfer confirmed - block: 14   gas used: 192451 (2.41%)
<Transaction object
↪'0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f'>
```

6.1.1 KYC Registrar

There are two types of investor registry contracts:

- KYCRegistrar.sol can be maintained by one or more authorities and used as a shared whitelist by many issuers
- KYCIssuer.sol is a more bare-bones registry, unique to a single issuer

Owner addresses are able to add investors to the registrar whitelist using `KYCRegistrar.addInvestor`.

```
>>> kyc = accounts[0].deploy(KYCRegistrar, [accounts[0]], 1)

Transaction sent: 0xd10264c1445aad4e9dc84e04615936624e0b96596fec2097bebc83f9d3e69664
KYCRegistrar.constructor confirmed - block: 2   gas used: 2853810 (35.67%)
KYCRegistrar deployed at: 0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06
<KYCRegistrar Contract object '0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06'>
>>>
>>> kyc.addInvestor("0x1234", 784, "0x465500", 2, 9999999999, (accounts[3],), {'from
↪': accounts[0]})

Transaction sent: 0x47581e5b276298427f6a520353622b96cdecb29dff7269f03d7c957435398ebd
KYCRegistrar.addInvestor confirmed - block: 3   gas used: 120707 (1.51%)
<Transaction object
↪'0x47581e5b276298427f6a520353622b96cdecb29dff7269f03d7c957435398ebd'>
```

See the *KYC* page for a detailed explanation of how to use registry contracts.

6.1.2 Issuing Tokens

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

1. Deploy IssuingEntity.sol.

```
>>> issuer = accounts[0].deploy(IssuingEntity, [accounts[0]], 1)

Transaction sent:␣
↪0xb37d8d16b266796e64fde6a4e9813ae0673dddaeb63022d91c706612ee741972
IssuingEntity.constructor confirmed - block: 2   gas used: 6473451 (80.92%)
IssuingEntity deployed at: 0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8
<IssuingEntity Contract object '0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8'>
```

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.

```
>>> issuer.setRegistrar(kyc, True, {'from': accounts[0]})

Transaction sent:␣
↪0x606326c8b2b8f1541c333ef5a5cd44592efb50530c6326e260e728095b3ec2bd
IssuingEntity.setRegistrar confirmed - block: 3   gas used: 61246 (0.77%)
<Transaction object
↪'0x606326c8b2b8f1541c333ef5a5cd44592efb50530c6326e260e728095b3ec2bd'>
```

3. Deploy SecurityToken.sol. Enter the address of the issuer contract from step one in the constructor. The authorized supply is set at deployment, the initial total supply will be zero.

```
>>> token = accounts[0].deploy(SecurityToken, issuer, "Test Token", "TST",␣
↪1000000)

Transaction sent:␣
↪0x4d2bbbc01d026de176bf5749e6e1bd22ba6eb40a225d2a71390f767b2845bacb
SecurityToken.constructor confirmed - block: 4   gas used: 3346083 (41.83%)
SecurityToken deployed at: 0x099c68D84815532A2C33e6382D6aD2C634E92ef6
<SecurityToken Contract object '0x099c68D84815532A2C33e6382D6aD2C634E92ef6'>
```

4. Call `IssuingEntity.addToken` to attach the token to the issuer.

```
>>> issuer.addToken(token, {'from': accounts[0]})

Transaction sent:␣
↪0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9
IssuingEntity.addToken confirmed - block: 5   gas used: 61630 (0.77%)
<Transaction object
↪'0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9'>
```

5. Call `IssuingEntity.setCountries` to approve investors from specific countries to hold the tokens.

```
>>> issuer.setCountries([784],[1],[0], {'from': accounts[0]})

Transaction sent:␣
↪0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3
IssuingEntity.setCountries confirmed - block: 6   gas used: 72379 (0.90%)
<Transaction object
↪'0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3'>
```

6. Call `SecurityToken.mint` to create new tokens, up to the authorized supply.

```
>>> token.mint(accounts[1], 1000, {'from': accounts[0]})

Transaction sent:␣
↪0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e
SecurityToken.mint confirmed - block: 13   gas used: 229092 (2.86%)
<Transaction object
↪'0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e'>
```

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.

6.1.3 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.

```
>>> token.checkTransfer(accounts[8], accounts[2], 500)
  File "/contract.py", line 277, in call
raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Address not␣
↪registered

>>> token.checkTransfer(accounts[1], accounts[2], 500)
True
```

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.

6.1.4 Custodians

There are many types of custodians possible. Included in the core SFT contracts is OwnedCustodian.sol, which is a basic implementation with a real-world owner.

Once a custodian contract is deployed you must attach it to an IssuingEntity with `IssuingEntity.addCustodian`.

```
>>> cust = accounts[0].deploy(OwnedCustodian, [accounts[0]], 1)

Transaction sent: 0x11540767a467504e3ddd03c8c2423840a69bd82a6f28db33ea869570b87486f0
OwnedCustodian.constructor confirmed - block: 13   gas used: 3326386 (41.58%)
OwnedCustodian deployed at: 0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D
<OwnedCustodian Contract object '0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D'>
>>>
>>> issuer.addCustodian(cust, {'from': accounts[0]})

Transaction sent: 0x63d13a81c73ed614ea68f1db8cc005bd860c6f2fb0ef7d590488672bd3edc5df
IssuingEntity.addCustodian confirmed - block: 14   gas used: 78510 (0.98%)
<Transaction object
↪'0x63d13a81c73ed614ea68f1db8cc005bd860c6f2fb0ef7d590488672bd3edc5df'>
```

At this point, transfers work in the following ways:

- Investors send tokens into the custodian contract just like they would any other address, using `SecurityToken.transfer` or `SecurityToken.transferFrom`.

```
>>> token.transfer(cust, 10000, {'from': accounts[1]})

Transaction sent:␣
↪0x4b09b29216d130dc06798ee673759a4e77e4823655c6477e895242f027726412
SecurityToken.transfer confirmed - block: 16   gas used: 155761 (1.95%)
<Transaction object
↪'0x4b09b29216d130dc06798ee673759a4e77e4823655c6477e895242f027726412'>
```

- Internal transfers within the custodian are done via `OwnedCustodian.transferInternal`.

```
>>> cust.transferInternal(token, accounts[1], accounts[2], 5000, {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x1c5cf1d01d2d5f9b9d9e801d8e2a0b9b2eb50fa11fbe03864b69ccf0fe2c03fc
OwnedCustodian.transferInternal confirmed - block: 17   gas used: 189610␣
↪(2.37%)
<Transaction object
↪'0x1c5cf1d01d2d5f9b9d9e801d8e2a0b9b2eb50fa11fbe03864b69ccf0fe2c03fc'>
```

- Transfers out of the custodian contract are initiated with `OwnedCustodian.transfer`.

```
>>> cust.transfer(token, accounts[2], 5000, {'from': accounts[0]})

Transaction sent:␣
↪0x227f7c24d68d63aa567c16458e039a283481ef5fd79d8b9e48c88b033ff18f79
OwnedCustodian.transfer confirmed - block: 18   gas used: 149638 (1.87%)
<Transaction object
↪'0x227f7c24d68d63aa567c16458e039a283481ef5fd79d8b9e48c88b033ff18f79'>
```

You can see an investor's custodied balance using `SecurityToken.custodianBalanceOf`.

```
>>> token.custodianBalanceOf(accounts[1], cust)
5000
```

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

6.2 Token

Each token contract represents a single class of securities from an issuer. Token contracts are based on the ERC20 Token Standard. Depending on the use case, there are two token implementations:

- SecurityToken.sol is used for the issuance of non-certificated (book entry) securities. These tokens are fungible.
- NFToken.sol is used for the issuance of certificated securities. These tokens are non-fungible.

Both contracts are derived from a common base Token.sol.

Token contracts include *MultiSig Implementation* and *Modules* via the associated *IssuingEntity* contract. 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.

Because of significant differences in the contracts, `SecurityToken` and `NFToken` are documented seperately.

6.2.1 SecurityToken

The `SecurityToken` contract represents a single class of fungible, non-certificated "book entry" securities. It is based on the ERC20 Token Standard, with an additional `checkTransfer` function available to verify if transfers will succeed.

Token contracts include *MultiSig Implementation* and *Modules* via the associated *IssuingEntity* contract. See the respective documents for more detailed information.

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

- SecurityToken.sol: the deployed contract, with functionality specific to `SecurityToken`.
- Token.sol: the base contract that both `SecurityToken` and `NFToken` inherit functionality from.

Deployment

`TokenBase.`**`constructor`**(*address _issuer*, *string _name*, *string _symbol*, *uint256 _authorizedSupply*)

- `_issuer`: The address of the `IssuingEntity` associated with this token.
- `_name`: The full name of the token.
- `_symbol`: The ticker symbol for the token.
- `_authorizedSupply`: The initial authorized token supply.

After the contract is deployed it must be associated with the issuer via `IssuingEntity.addToken`. It is not possible to mint tokens until this is done.

At the time of deployment the initial authorized supply is set, and the total supply is left as 0. The issuer may then mint tokens by calling `SecurityToken.mint` directly or via a module. See *Total Supply, Minting and Burning*.

```
>>> token = accounts[0].deploy(SecurityToken, issuer, "Test Token", "TST",␣
↪1000000)

Transaction sent:␣
↪0x4d2bbbc01d026de176bf5749e6e1bd22ba6eb40a225d2a71390f767b2845bacb
SecurityToken.constructor confirmed - block: 4   gas used: 3346083 (41.83%)
SecurityToken deployed at: 0x099c68D84815532A2C33e6382D6aD2C634E92ef6
<SecurityToken Contract object '0x099c68D84815532A2C33e6382D6aD2C634E92ef6'>
```

Public Constants

The following public variables cannot be changed after contract deployment.

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

```
>>> token.name()
Test Token
```

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

```
>>> token.symbol()
TST
```

TokenBase.**decimals**()
The number of decimal places for the token. In the standard SFT implementation this is set to 0.

```
>>> token.decimals()
0
```

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

```
>>> token.ownerID()
0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63
```

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

```
>>> token.issuer()
0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06
```

Total Supply, Minting and Burning

Authorized Supply

Along with the ERC20 standard `totalSupply`, token contracts include an `authorizedSupply` that represents the maximum allowable total supply. The issuer may mint new tokens using `SecurityToken.mint` until the total supply is equal to the authorized supply. The initial authorized supply is set during deployment and may be increased later using `TokenBase.modifyAuthorizedSupply`.

A *Governance* module can be deployed to dictate when the issuer is allowed to modify the authorized supply.

TokenBase.**modifyAuthorizedSupply**(*uint256 _value*)
Sets the authorized supply. The value may never be less than the current total supply.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

If a *Governance* module has been set on the associated `IssuingEntity`, it must provide approval whenever this method is called.

Emits the `AuthorizedSupplyChanged` event.

```
>>> token.modifyAuthorizedSupply(2000000, {'from': accounts[0]})

Transaction sent:␣
↪0x83b7a23e1bc1248445b64f275433add538f05336a4fe07007d39edbd06e1f476
SecurityToken.modifyAuthorizedSupply confirmed - block: 13   gas used: 46666 (0.58
↪%)
<Transaction object
↪'0x83b7a23e1bc1248445b64f275433add538f05336a4fe07007d39edbd06e1f476'>
```

Minting and Burning

SecurityToken.**mint**(*address _owner*, *uint256 _value*)
Mints new tokens at the given address.

- `_owner`: Account balance to mint tokens to.

- `_value`: Number of tokens to mint.

A `Transfer` even will fire showing the new tokens as transferring from `0x00` and the total supply will increase. The new total supply cannot exceed `authorizedSupply`.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

Modules can hook into this method via `STModule.totalSupplyChanged`.

```
>>> token.mint(accounts[1], 5000, {'from': accounts[0]})

Transaction sent:␣
↪0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e
SecurityToken.mint confirmed - block: 14   gas used: 229092 (2.86%)
<Transaction object
↪'0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e'>
```

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

Burns tokens at the given address.

- `_owner`: Account balance to burn tokens from.
- `_value`: Number of tokens to burn.

A `Transfer` event is emitted showing the new tokens as transferring to `0x00` and the total supply will increase.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

Modules can hook into this method via `STModule.totalSupplyChanged`.

```
>>> token.burn(accounts[1], 1000, {'from': accounts[0]})

Transaction sent:␣
↪0x5414b31e3e44e657ed5ee04c0c6e4c673ab2c6300f392dfd7c282b348db0bbc7
SecurityToken.burn confirmed - block: 15   gas used: 48312 (0.60%)
<Transaction object
↪'0x5414b31e3e44e657ed5ee04c0c6e4c673ab2c6300f392dfd7c282b348db0bbc7'>
```

Getters

`TokenBase.`**`totalSupply`**`()`

Returns the current total supply of tokens.

```
>>> token.totalSupply()
5000
```

`TokenBase.`**`authorizedSupply`**`()`

Returns the maximum authorized total supply of tokens. Whenever the authorized supply exceeds the total supply, the issuer may mint new tokens using `SecurityToken.mint`.

```
>>> token.authorizedSupply()
2000000
```

`TokenBase.`**`treasurySupply`**`()`

Returns the number of tokens held by the issuer. Equivalent to calling `TokenBase.balanceOf(issuer)`.

```
>>> token.treasurySupply()
1000
>>> token.balanceOf(issuer)
1000
```

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

```
>>> token.circulatingSupply()
4000
```

Balances and Transfers

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

Checking Balances

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

```
>>> token.balanceOf(accounts[1])
4000
```

TokenBase.**custodianBalanceOf**(*address _owner*, *address _cust*)
Returns the custodied token balance for a given address.

```
>>> token.custodianBalanceOf(accounts[1], cust)
0
```

TokenBase.**allowance**(*address _owner*, *address _spender*)
Returns the amount of tokens that `_spender` may transfer from `_owner`'s balance using `SecurityToken.transferFrom`.

```
>>> token.allowance(accounts[1], accounts[2])
1000
```

Checking Transfer Permissions

TokenBase.**checkTransfer**(*address _from*, *address _to*, *uint256 _value*)
Checks if a token transfer is permitted.

- `_from`: Address of the sender
- `_to`: Address of the recipient
- `_value`: Amount of tokens to be transferred

Returns `true` if the transfer is permitted. If the transfer is not permitted, the call will revert with the reason given in the error string.

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.

Modules can hook into this method via `STModule.checkTransfer`.

```
>>> token.checkTransfer(accounts[1], accounts[2], 100)
True
>>> token.checkTransfer(accounts[1], accounts[2], 10000)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert␣
↪Insufficient Balance
>>> token.checkTransfer(accounts[1], accounts[9], 100)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Address␣
↪not registered
```

`TokenBase.`**`checkTransferCustodian`**(*address _cust*, *address _from*, *address _to*, *uint256 _value*)

Checks if a custodian internal transfer of tokens is permitted. See the *Custodians* documentation for more information on custodial internal transfers.

- `_cust`: Address of the custodian
- `_from`: Address of the sender
- `_to`: Address of the recipient
- `_value`: Amount of tokens to be transferred

Returns `true` if the transfer is permitted. If the transfer is not permitted, the call will revert with the reason given in the error string.

Permissioning checks for custodial transfers are identical to those of normal transfers.

Modules can hook into this method via `STModule.checkTransfer`. A custodial transfer can be differentiated from a regular transfer because the caller ID is be that of the custodian.

```
>>> token.custodianBalanceOf(accounts[1], cust)
2000
>>> token.checkTransferCustodian(cust, accounts[1], accounts[2], 1000)
True
>>> token.checkTransferCustodian(cust, accounts[1], accounts[2], 5000)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert␣
↪Insufficient Custodial Balance
```

Transferring Tokens

`SecurityToken.`**`transfer`**(*address _to*, *uint256 _value*)

Transfers `_value` tokens from `msg.sender` to `_to`. If the transfer cannot be completed, the call will revert with the reason given in the error string.

Some logic in this method deviates from the ERC20 standard, see *Non Standard Behaviours* for more information.

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

```
>>> token.transfer(accounts[2], 1000, {'from': accounts[1]})

Transaction sent:␣
↪0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f
SecurityToken.transfer confirmed - block: 14   gas used: 192451 (2.41%)
<Transaction object
↪'0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f'>
```

`TokenBase.`**`approve`**(*address _spender*, *uint256 _value*)

Approves `_spender` to transfer up to `_value` tokens belonging to `msg.sender`.

If `_spender` is already approved for >0 tokens, the caller must first set approval to 0 before setting a new value. This prevents the attack vector documented here.

No transfer permission logic is applied when making this call. 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.

Emits the `Approval` event.

```
>>> token.approve(accounts[2], 1000, {'from': accounts[1]})

Transaction sent:␣
↪0xa8793d57cfbf6e6ed0507c62e09c31c34feaae503b69aa6e6f4d39fad36fd7c5
SecurityToken.approve confirmed - block: 20   gas used: 45948 (0.57%)
<Transaction object
↪'0xa8793d57cfbf6e6ed0507c62e09c31c34feaae503b69aa6e6f4d39fad36fd7c5'>
```

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

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

Prior approval must have been given via `TokenBase.approve`, except in certain cases documented under *Non Standard Behaviours*.

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

Modules can hook into this method via `STModule.transferTokens`.

```
>>> token.transferFrom(accounts[1], accounts[3], 1000, {'from': accounts[2]})

Transaction sent:␣
↪0x84cdd0c85d3e39f1ba4f5cbd0c4cb196c0f343c90c0819157acd14f6041fe945
SecurityToken.transferFrom confirmed - block: 21   gas used: 234557 (2.93%)
<Transaction object
↪'0x84cdd0c85d3e39f1ba4f5cbd0c4cb196c0f343c90c0819157acd14f6041fe945'>
```

Modules

Modules are attached and detached to token contracts via the associated `IssuingEntity`. See *Attaching and Detaching*.

`TokenBase.`**`isActiveModule`**(*address _module*)
Returns `true` if a module is currently active on the token. Modules that are active on the associated `IssuingEntity` are also considered active on tokens. If the module is not active, returns `false`.

```
>>> token.isActiveModule(token_module)
True
>>> token.isActiveModule(other_module)
False
```

`TokenBase.`**`isPermittedModule`**(*address _module*, *bytes4 _sig*)
Returns `true` if a module is permitted to access a specific method. If the module is not active or not permitted to call the method, returns `false`.

```
>>> token.isPermittedModule(token_module, "0x40c10f19")
True
>>> token.isPermittedModule(token_module, "0xc39f42ed")
False
```

Events

The `SecurityToken` contract includes the following events.

`TokenBase.`**`Transfer`**(*address indexed from*, *address indexed to*, *uint256 tokens*)
Emitted when a token transfer is completed via `SecurityToken.transfer` or `SecurityToken.transferFrom`.

Also emitted by `SecurityToken.mint` and `SecurityToken.burn`. For minting the address of the sender will be `0x00`, for burning it will be the address of the receiver.

`TokenBase.`**`Approval`**(*address indexed tokenOwner*, *address indexed spender*, *uint256 tokens*)
Emitted when an approved transfer amount is set via `SecurityToken.approve`.

`TokenBase.`**`AuthorizedSupplyChanged`**(*uint256 oldAuthorized*, *uint256 newAuthorized*)
Emitted when the authorized supply is changed via `TokenBase.modifyAuthorizedSupply`.

6.2.2 NFToken

The `NFToken` contract represents a single class of non-fungible certificated securities. It is based on the ERC20 Token Standard, however it introduces significant additional functionality to allow full non-fungibility of tokens at scale.

Token contracts include *MultiSig Implementation* and *Modules* via the associated *IssuingEntity* contract. See the respective documents for more detailed information.

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

- NFToken.sol: the deployed contract, with functionality specific to `NFToken`.
- Token.sol: the base contract that both `NFToken` and `SecurityToken` inherit functionality from.

How it Works

`NFToken` applies a unique, sequential index value to every token. This results in fully non-fungible tokens that can transfer at scale without prohibitively high gas costs.

The first token minted will have an index value of 1. The maximum index value is 281474976710654 (`2**48 - 2`). References to token ranges are in the format `start:stop` where the final included value is `stop-1`. For example, a range of `2:6` would contains tokens 2, 3, 4 and 5.

Each range includes the following values:

- `_time`: A `uint32` epoch time based transfer restriction that is applied to the range. The tokens cannot be transferred until `now > _time`. Maximum value is 4294967295 (February, 2106).
- `_tag`: A `bytes2` tag attached to the range, that allows for more granular control over which modules are called when attempting to transfer the range. See *Hooks and Tags* for more information.

These values are initially set at the time of minting and can be modified later with `NFToken.modifyRange` or `NFToken.modifyRanges`. See *Token Ranges* for more information on these methods.

Any time a range is created, modified or transferred, the contract will merge it with neighboring ranges if possible.

To track the chain of custody for each token, monitor the `TransferRange` event.

Deployment

`TokenBase.`**`constructor`**(*address _issuer*, *string _name*, *string _symbol*, *uint256 _authorizedSupply*)

- `_issuer`: The address of the `IssuingEntity` associated with this token.
- `_name`: The full name of the token.
- `_symbol`: The ticker symbol for the token.
- `_authorizedSupply`: The initial authorized token supply.

After the contract is deployed it must be associated with the issuer via `IssuingEntity.addToken`. It is not possible to mint tokens until this is done.

At the time of deployment the initial authorized supply is set, and the total supply is left as 0. The issuer may then mint tokens by calling `NFToken.mint` directly or via a module. See *Total Supply, Minting and Burning*.

```
>>> token = accounts[0].deploy(NFToken, issuer, "Test Token", "TST", 1000000)

Transaction sent:␣
↪0x4d2bbbc01d026de176bf5749e6e1bd22ba6eb40a225d2a71390f767b2845bacb
NFToken.constructor confirmed - block: 4   gas used: 3346083 (41.83%)
NFToken deployed at: 0x099c68D84815532A2C33e6382D6aD2C634E92ef6
<NFToken Contract object '0x099c68D84815532A2C33e6382D6aD2C634E92ef6'>
```

Public Constants

The following public variables cannot be changed after contract deployment.

`TokenBase.`**`name`**`()`

The full name of the security token.

```
>>> token.name()
Test Token
```

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

```
>>> token.symbol()
TST
```

`TokenBase.`**`decimals`**`()`
The number of decimal places for the token. In the standard SFT implementation this is set to 0.

```
>>> token.decimals()
0
```

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

```
>>> token.ownerID()
0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63
```

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

```
>>> token.issuer()
0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06
```

Total Supply, Minting and Burning

Authorized Supply

Along with the ERC20 standard `totalSupply`, token contracts include an `authorizedSupply` that represents the maximum allowable total supply. The issuer may mint new tokens using `NFToken.mint` until the total supply is equal to the authorized supply. The initial authorized supply is set during deployment and may be increased later using `TokenBase.modifyAuthorizedSupply`.

A *Governance* module can be deployed to dictate when the issuer is allowed to modify the authorized supply.

`TokenBase.`**`modifyAuthorizedSupply`**`(`*uint256 _value*`)`
Sets the authorized supply. The value may never be less than the current total supply.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

If a *Governance* module has been set on the associated `IssuingEntity`, it must provide approval whenever this method is called.

Emits the `AuthorizedSupplyChanged` event.

```
>>> token.modifyAuthorizedSupply(2000000, {'from': accounts[0]})

Transaction sent:␣
↪0x83b7a23e1bc1248445b64f275433add538f05336a4fe07007d39edbd06e1f476
NFToken.modifyAuthorizedSupply confirmed - block: 13   gas used: 46666 (0.58%)
<Transaction object
↪'0x83b7a23e1bc1248445b64f275433add538f05336a4fe07007d39edbd06e1f476'>
```

Minting and Burning

`NFToken.`**`mint`**(*address _owner*, *uint48 _value*, *uint32 _time*, *bytes2 _tag*)

Mints new tokens at the given address.

- `_owner`: Account balance to mint tokens to.
- `_value`: Number of tokens to mint.
- `_time`: Time restriction to apply to tokens.
- `_tag`: Tag to apply to tokens.

A `Transfer` even will fire showing the new tokens as transferring from `0x00` and the total supply will increase. The new total supply cannot exceed `authorizedSupply` and the upper bound of the range cannot exceed `2**48 - 2`.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

Modules can hook into this method via `STModule.totalSupplyChanged`.

```
>>> token.mint(accounts[1], 5000, 0, "0x0000", {'from': accounts[0]})

Transaction sent:␣
↪0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e
NFToken.mint confirmed - block: 14   gas used: 229092 (2.86%)
<Transaction object
↪'0x77ec76224d90763641971cd61e99711c911828053612cc16eb2e5d7faa20815e'>
```

`NFToken.`**`burn`**(*uint48 _start*, *uint48 _stop*)

Burns tokens at the given range.

- `_start`: Start index of token range to burn.
- `_stop`: Stop index of token range to burn.

Burning a partial range is allowed. Burning tokens from multiple ranges in the same call is not. Once tokens are burnt they are gone forever, their index values will never be re-used.

A `Transfer` event is emitted showing the new tokens as transferring to `0x00` and the total supply will increase.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

Modules can hook into this method via `STModule.totalSupplyChanged`.

```
>>> token.burn(accounts[1], 1000, {'from': accounts[0]})

Transaction sent:␣
↪0x5414b31e3e44e657ed5ee04c0c6e4c673ab2c6300f392dfd7c282b348db0bbc7
NFToken.burn confirmed - block: 15   gas used: 48312 (0.60%)
<Transaction object
↪'0x5414b31e3e44e657ed5ee04c0c6e4c673ab2c6300f392dfd7c282b348db0bbc7'>
```

Getters

`TokenBase.`**`totalSupply`**()

Returns the current total supply of tokens.

```
>>> token.totalSupply()
5000
```

TokenBase.**authorizedSupply**()
Returns the maximum authorized total supply of tokens. Whenever the authorized supply exceeds the total supply, the issuer may mint new tokens using NFToken.mint.

```
>>> token.authorizedSupply()
2000000
```

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

```
>>> token.treasurySupply()
1000
>>> token.balanceOf(issuer)
1000
```

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

```
>>> token.circulatingSupply()
4000
```

Token Ranges

If you haven't yet, read the *How it Works* section for an introduction to how token ranges work within this contract.

Modifying Ranges

NFToken.**modifyRange**(*uint48 _pointer*, *uint32 _time*, *bytes2 _tag*)
Modifies the time restriction and tag for a single range.

- _pointer: Start index of the range to modify
- _time: New time restriction for the range
- _tag: New tag for the range

If the index given in _pointer is not the first token in a range, the call will revert.

This method is callable directly by the issuer, implementing multi-sig via MultiSig.checkMultiSigExternal. It may also be called by a permitted module.

Emits the RangeSet event.

```
>>> token.getRange(1).dict()
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0xf414d65808f5f59aE156E51B97f98094888e7d92",
    '_start': 1,
    '_stop': 1000,
    '_tag': "0x0000",
    '_time': 0
}
>>> token.modifyRange(1, 1600000000, "0x1234", {'from':accounts[0]})
```

(continues on next page)

(continued from previous page)

```
Transaction sent:␣
↪0xed36d04d4888db5d9fefb69b0fa98367f19049d304f60c55b6a1b74da3fd8edd
NFToken.modifyRange confirmed - block: 18   gas used: 51594 (0.64%)
>>> token.getRange(1).dict()
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0xf414d65808f5f59aE156E51B97f98094888e7d92",
    '_start': 1,
    '_stop': 1000,
    '_tag': "0x1234",
    '_time': 1600000000
}
```

`NFToken.`**`modifyRanges`**(*uint48 _start*, *uint48 _stop*, *uint32 _time*, *bytes2 _tag*)

Modifies the time restriction and tag for all tokens within a given range.

- `_start`: Start index of the range to modify
- `_stop`: Stop index of the range to modify.
- `_time`: New time restriction for the range
- `_tag`: New tag for the range

This method may be used to apply changes across multiple ranges, or to modify a portion of a single range.

This method is callable directly by the issuer, implementing multi-sig via `MultiSig.checkMultiSigExternal`. It may also be called by a permitted module.

Emits the `RangeSet` event for each range that is modified.

```
>>> token.getRange(1).dict()
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0xf414d65808f5f59aE156E51B97f98094888e7d92",
    '_start': 1,
    '_stop': 1000,
    '_tag': "0x0000",
    '_time': 0
}
>>> token.modifyRanges(500, 1500, 2000000000, "0xffff", {'from':accounts[0]})

Transaction sent:␣
↪0xe9a6d2e961bdd24339d24c140e8d16fd69cf93a72fc93810798aa0d2bbe69525
NFToken.modifyRanges confirmed - block: 21   gas used: 438078 (5.48%)
<Transaction object
↪'0xe9a6d2e961bdd24339d24c140e8d16fd69cf93a72fc93810798aa0d2bbe69525'>
>>>
>>> token.getRange(1).dict()
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0xf414d65808f5f59aE156E51B97f98094888e7d92",
    '_start': 1,
    '_stop': 500,
    '_tag': "0x0000",
    '_time': 0
}
>>> token.getRange(500).dict()
```

(continues on next page)

(continued from previous page)

```
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0xf414d65808f5f59aE156E51B97f98094888e7d92",
    '_start': 500,
    '_stop': 1000,
    '_tag': "0xffff",
    '_time': 2000000000
}
```

Getters

References to token ranges are in the format `start:stop` where the final included value is `stop-1`. For example, a range of `2:6` would contains tokens 2, 3, 4 and 5.

`NFToken.`**`getRange`**`(`*uint256 _idx*`)`

Returns information about the token range that `_idx` is a part of.

```
>>> token.getRange(1337).dict()
{
    '_custodian': "0x0000000000000000000000000000000000000000",
    '_owner': "0x055f1c2c9334a4e57ACF2C4d7ff95d03CA7d6741",
    '_start': 1000,
    '_stop': 2000,
    '_tag': "0x0000",
    '_time': 0
}
```

`NFToken.`**`rangesOf`**`(`*address _owner*`)`

Returns the `start:stop` indexes of each token range belonging to `_owner`.

```
>>> token.rangesOf(accounts[1])
((1, 1000), (2000, 10001))
```

`NFToken.`**`custodianRangesOf`**`(`*address _owner*, *address _custodian*`)`

Returns the `start:stop` indexes of each token range belonging to `_owner` that is custodied by `_custodian`.

```
>>> token.custodianRangesOf(accounts[1], cust)
((1000, 2000))
```

Balances and Transfers

NFToken includes the standard ERC20 methods for token transfers, however their functionality differs slightly due to transfer permissioning requirements. It also introduces new methods to allow finer control around transfer of specific token ranges.

Checking Balances

`TokenBase.`**`balanceOf`**`(`*address*`)`

Returns the token balance for a given address.

```
>>> token.balanceOf(accounts[1])
4000
```

TokenBase.**custodianBalanceOf**(*address _owner*, *address _cust*)

Returns the custodied token balance for a given address.

```
>>> token.custodianBalanceOf(accounts[1], cust)
0
```

TokenBase.**allowance**(*address _owner*, *address _spender*)

Returns the amount of tokens that `_spender` may transfer from `_owner`'s balance using `NFToken.transferFrom`.

```
>>> token.allowance(accounts[1], accounts[2])
1000
```

Checking Transfer Permissions

TokenBase.**checkTransfer**(*address _from*, *address _to*, *uint256 _value*)

Checks if a token transfer is permitted.

- `_from`: Address of the sender
- `_to`: Address of the recipient
- `_value`: Amount of tokens to be transferred

Returns `true` if the transfer is permitted. If the transfer is not permitted, the call will revert with the reason given in the error string.

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.

Modules can hook into this method via `STModule.checkTransfer`.

```
>>> token.checkTransfer(accounts[1], accounts[2], 100)
True
>>> token.checkTransfer(accounts[1], accounts[2], 10000)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert␣
↪Insufficient Balance
>>> token.checkTransfer(accounts[1], accounts[9], 100)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Address␣
↪not registered
```

(continues on next page)

(continued from previous page)

TokenBase.**checkTransferCustodian**(*address _cust*, *address _from*, *address _to*, *uint256 _value*)

Checks if a custodian internal transfer of tokens is permitted. See the *Custodians* documentation for more information on custodial internal transfers.

- `_cust`: Address of the custodian
- `_from`: Address of the sender
- `_to`: Address of the recipient
- `_value`: Amount of tokens to be transferred

Returns `true` if the transfer is permitted. If the transfer is not permitted, the call will revert with the reason given in the error string.

Permissioning checks for custodial transfers are identical to those of normal transfers.

Modules can hook into this method via `STModule.checkTransfer`. A custodial transfer can be differentiated from a regular transfer because the caller ID is be that of the custodian.

```
>>> token.custodianBalanceOf(accounts[1], cust)
2000
>>> token.checkTransferCustodian(cust, accounts[1], accounts[2], 1000)
True
>>> token.checkTransferCustodian(cust, accounts[1], accounts[2], 5000)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert␣
↪Insufficient Custodial Balance
```

Transferring Tokens

NFToken.**transfer**(*address _to*, *uint256 _value*)

Transfers `_value` tokens from `msg.sender` to `_to`. If the transfer cannot be completed, the call will revert with the reason given in the error string.

This call will iterate through each range owned by the caller and transfer them until `_value` tokens have been sent. If a partial range is sent, it will split it and send the range with a lower start index. For example, if the sender owns range `1000:2000` and `_value` is 400 tokens, it will transfer `1000:1400` to the receiver.

Some logic in this method deviates from the ERC20 standard, see *Non Standard Behaviours* for more information.

All transfers will emit the `Transfer` event, as well as one or more `TransferRange` events. Transfers where there is a change of ownership will also emit‘‘IssuingEntity.TransferOwnership‘‘.

```
>>> token.transfer(accounts[2], 1000, {'from': accounts[1]})

Transaction sent:␣
↪0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f
NFToken.transfer confirmed - block: 14   gas used: 192451 (2.41%)
<Transaction object
↪'0x29d9786ca39e79714581b217c24593546672e31dbe77c64804ea2d81848f053f'>
```

TokenBase.**approve**(*address _spender*, *uint256 _value*)

Approves `_spender` to transfer up to `_value` tokens belonging to `msg.sender`.

If `_spender` is already approved for >0 tokens, the caller must first set approval to 0 before setting a new value. This prevents the attack vector documented here.

No transfer permission logic is applied when making this call. 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.

Emits the `Approval` event.

```
>>> token.approve(accounts[2], 1000, {'from': accounts[1]})

Transaction sent:␣
↪0xa8793d57cfbf6e6ed0507c62e09c31c34feaae503b69aa6e6f4d39fad36fd7c5
NFToken.approve confirmed - block: 20   gas used: 45948 (0.57%)
<Transaction object
↪'0xa8793d57cfbf6e6ed0507c62e09c31c34feaae503b69aa6e6f4d39fad36fd7c5'>
```

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

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

Prior approval must have been given via `TokenBase.approve`, except in certain cases documented under *Non Standard Behaviours*.

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

Modules can hook into this method via `STModule.transferTokens`.

```
>>> token.transferFrom(accounts[1], accounts[3], 1000, {'from': accounts[2]})

Transaction sent:␣
↪0x84cdd0c85d3e39f1ba4f5cbd0c4cb196c0f343c90c0819157acd14f6041fe945
NFToken.transferFrom confirmed - block: 21   gas used: 234557 (2.93%)
<Transaction object
↪'0x84cdd0c85d3e39f1ba4f5cbd0c4cb196c0f343c90c0819157acd14f6041fe945'>
```

`NFToken.`**`transferRange`**(*address _to*, *uint48 _start*, *uint48 _stop*)

Transfers the token range `_start:_stop` from `msg.sender` to `_to`.

Transferring a partial range is allowed. Transferring tokens from multiple ranges in the same call is not.

All transfers will emit the `Transfer` and `TransferRange` events. Transfers where there is a change of ownership will also emit‘‘IssuingEntity.TransferOwnership‘‘.

```
>>> token.transferRange(accounts[2], 1000, 2000, {'from': accounts[1]})

Transaction sent:␣
↪0x9ae3c41984aad767b2a535a5ade8f70b104b125da622124e9c3be52b7e373a11
NFToken.transferRange confirmed - block: 17   gas used: 441081 (5.51%)
<Transaction object
↪'0x9ae3c41984aad767b2a535a5ade8f70b104b125da622124e9c3be52b7e373a11'>
```

Modules

Modules are attached and detached to token contracts via the associated `IssuingEntity`. See *Attaching and Detaching*.

`TokenBase.`**`isActiveModule`**(*address _module*)
Returns `true` if a module is currently active on the token. Modules that are active on the associated `IssuingEntity` are also considered active on tokens. If the module is not active, returns `false`.

```
>>> token.isActiveModule(token_module)
True
>>> token.isActiveModule(issuer_module)
True
```

`TokenBase.`**`isPermittedModule`**(*address _module*, *bytes4 _sig*)
Returns `true` if a module is permitted to access a specific method. If the module is not active or not permitted to call the method, returns `false`.

```
>>> token.isPermittedModule(token_module, "0x40c10f19")
True
>>> token.isPermittedModule(token_module, "0xc39f42ed")
False
```

Events

The `NFToken` contract includes the following events.

`TokenBase.`**`Transfer`**(*address indexed from*, *address indexed to*, *uint256 tokens*)
Emitted when a token transfer is completed via `NFToken.transfer` or `NFToken.transferFrom`.

Also emitted by `NFToken.mint` and `NFToken.burn`. For minting the address of the sender will be `0x00`, for burning it will be the address of the receiver.

`NFToken.`**`TransferRange`**(*address indexed from*, *address indexed to*, *uint256 start*, *uint256 stop*, *uint256 amount*)
Emitted whenever a token range is transferred via `NFToken.transferRange`.

Emitted once for each range transferred during calls to `NFToken.transfer` and `NFToken.transferFrom`.

Also emitted by `NFToken.mint` and `NFToken.burn`. For minting the address of the sender will be `0x00`, for burning it will be the address of the receiver.

`TokenBase.`**`Approval`**(*address indexed tokenOwner*, *address indexed spender*, *uint256 tokens*)
Emitted when an approved transfer amount is set via `NFToken.approve`.

`TokenBase.`**`AuthorizedSupplyChanged`**(*uint256 oldAuthorized*, *uint256 newAuthorized*)
Emitted when the authorized supply is changed via `TokenBase.modifyAuthorizedSupply`.

`NFToken.`**`RangeSet`**(*bytes2 indexed tag*, *uint256 start*, *uint256 stop*, *uint32 time*)
Emitted when a token range is modified via `NFToken.modifyRange` or `NFToken.modifyRanges`, or when a new range is minted with `NFToken.mint`.

6.2.3 Non Standard Behaviours

`SecurityToken` and `NFToken` are based upon the ERC20 Token Standard, however they deviate in several areas.

Issuer Balances

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

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

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

Token Transfers

The following behaviours deviate from ERC20 relating to token transfers:

- Transfers of 0 tokens will revert with an error string "Cannot send 0 tokens".
- If the caller and sender addresses are both associated to the same ID, `TokenBase.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.
- The issuer may call `TokenBase.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.

6.3 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 multis-gi functionality. See *MultiSig Implementation* for detailed information on this component.

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.

6.3.1 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`.

```
>>> issuer = accounts[0].deploy(IssuingEntity, [accounts[0], accounts[1]], 1)

Transaction sent:␣
↪0xb37d8d16b266796e64fde6a4e9813ae0673dddaeb63022d91c706612ee741972
IssuingEntity.constructor confirmed - block: 1   gas used: 6473451 (80.92%)
IssuingEntity deployed at: 0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8
<IssuingEntity Contract object '0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8'>
```

6.3.2 Public Constants

The following public variables cannot be changed after contract deployment.

`IssuingEntity.`**`ownerID`**`()`
The bytes32 ID hash of the issuer.

```
>>> issuer.ownerID()
0xce1e12589ad8fb3eed11af5b9ef8788c25b574d4073d23c871e003021400c429
```

6.3.3 Tokens, Registrars, Custodians, Governance

The `IssuingEntity` contract is a center point through which other contracts are linked. Each contract must be associated to it before it will function properly.

- *SecurityToken* contracts must be associated before tokens can be transferred, so that the issuer contract can accurately track investor counts.
- *KYC* contracts must be associated to provide KYC data on investors before they can receive or send tokens.
- *Custodians* contracts must be approved in order to send or receive tokens from investors.
- A *Governance* contract may optionally be associated. Once attached, it requires the issuer to receive on-chain approval before creating or minting additional tokens.

Associating Contracts

`IssuingEntity.`**`addToken`**`(`*address _token*`)`
Associates a new *SecurityToken* contract with the issuer contract.

Once added, the token can be restricted with `IssuingEntity.setTokenRestriction`.

If a *Governance* module has been set, it must provide approval whenever this method is called.

Emits the `TokenAdded` event.

```
>>> issuer.addToken(SecurityToken[0], {'from': accounts[0]})

Transaction sent:␣
↪0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9
IssuingEntity.addToken confirmed - block: 5   gas used: 61630 (0.77%)
<Transaction object
↪'0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9'>
```

`IssuingEntity.`**`setRegistrar`**`(`*address _registrar*, *bool _restricted*`)`
Associates or removes a *KYC* contract.

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".

Emits the `RegistrarSet` event.

```
>>> issuer.setRegistrar(KYCRegistrar[0], False, {'from': accounts[0]})

Transaction sent:␣
↪0x606326c8b2b8f1541c333ef5a5cd44592efb50530c6326e260e728095b3ec2bd
IssuingEntity.setRegistrar confirmed - block: 3   gas used: 61246 (0.77%)
<Transaction object
↪'0x606326c8b2b8f1541c333ef5a5cd44592efb50530c6326e260e728095b3ec2bd'>
```

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

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

Once a custodian has been added, they can be restricted with `IssuingEntity.setEntityRestriction`.

Emits the `CustodianAdded` event.

```
>>> issuer.addCustodian(OwnedCustodian[0])

Transaction sent:␣
↪0xbae451ce98691dc37dad6a67d8daf410a3eeebf34b59ab60eaeef7c3f3a2654c
IssuingEntity.addCustodian confirmed - block: 25   gas used: 78510 (0.98%)
<Transaction object
↪'0xbae451ce98691dc37dad6a67d8daf410a3eeebf34b59ab60eaeef7c3f3a2654c'>
```

`IssuingEntity.`**`setGovernance`**(*address _governance*)

Sets the active *Governance* contract.

Setting the address to `0x00` disables governance functionality.

```
>>> issuer.setGovernance(GovernanceMinimal[0])

Transaction sent:␣
↪0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9
IssuingEntity.addCustodian confirmed - block: 26   gas used: 63182 (0.98%)
<Transaction object
↪'0x8e93cd6b85d1e993755e9fe31eb14ce600706eaf98d606156447d8e431db5db9'>
```

Setting Restrictions

Transfer restrictions can be applied at varying levels.

`IssuingEntity.`**`setEntityRestriction`**(*bytes32 _id*, *bool _restricted*)

Retricts or permits an investor or custodian 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.

Emits the `EntityRestriction` event.

```
>>> SecurityToken[0].transfer(accounts[2], 100, {'from': accounts[1]})

Transaction sent:␣
↪0x89bf6113bd5ccf9917d0749776fa4bed986d519d66221973def33c0190a2e6d2
SecurityToken.transfer confirmed - block: 21   gas used: 192387 (2.40%)
>>> issuer.setEntityRestriction(id_, True)
```

(continues on next page)

(continued from previous page)

```
Transaction sent:␣
↪0xfc4dabf2c48b4502ab4a9d3edbfc0ea792e715069ede0f8b455697df180bfc9f
IssuingEntity.setEntityRestriction confirmed - block: 22   gas used: 39978 (0.50%)
>>> SecurityToken[0].transfer(accounts[2], 100, {'from': accounts[1]})
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Sender␣
↪restricted: Issuer
```

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

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

Emits the `TokenRestriction` event.

```
>>> issuer.setTokenRestriction(SecurityToken[0], True, {'from': accounts[0]})

Transaction sent:␣
↪0xfe60d18d0315278bdd1cfd0896a040cdadb63ada255685737908672c0cd10cee
IssuingEntity.setTokenRestriction confirmed - block: 13   gas used: 40369 (0.50%)
<Transaction object
↪'0xfe60d18d0315278bdd1cfd0896a040cdadb63ada255685737908672c0cd10cee'>
```

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

Restricts or permits transfers of all associated tokens. Modifying the global restriction does not affect individual token restrictions - i.e. you cannot call this method to remove restrictions that were set with `IssuingEntity.setTokenRestriction`.

Emits the `GlobalRestriction` event.

```
>>> issuer.setGlobalRestriction(True, {'from': accounts[0]})

Transaction sent:␣
↪0xc03ac4c6d36e971f980297e365f30752ac5097e391213c59fd52544829a87479
IssuingEntity.setGlobalRestriction confirmed - block: 14   gas used: 53384 (0.67%)
<Transaction object
↪'0xc03ac4c6d36e971f980297e365f30752ac5097e391213c59fd52544829a87479'>
```

Getters

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

Returns a boolean indicating if the given address is a token contract that is associated with the `IssuingEntity` not currently restricted.

```
>>> issuer.isActiveToken(SecurityToken[0])
True
>>> issuer.isActiveToken(accounts[2])
False
```

`IssuingEntity.`**`governance`**()

Returns the address of the associated `Governance` contract. If none is set, returns `0x00`.

```
>>> issuer.governance()
"0x14b0Ed2a7C4cC60DD8F676AE44D0831d3c9b2a9E"
```

6.3.4 Investors

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

Getters

The `IssuingEntity` contract contains several public getter methods for querying information relating to investors.

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

Check if an address belongs to a registered investor and return a bool. Returns `false` if the address is not registered.

```
>>> issuer.isRegisteredInvestor(accoounts[2])
True
>>> issuer.isRegisteredInvestor(accoounts[9])
False
```

`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. If the ID cannot be found the call will revert.

```
>>> issuer.getID(accounts[1])
0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63
>>> issuer.getID(accounts[9])
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Address␣
↪not registered
```

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

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

Note that an investor's ID is only saved in the contract after a successful token transfer. Even if the investor's ID is known via an associated registrar, if they have never received tokens the call to `getInvestorRegistrar` will return an empty value.

```
>>> id_ = issuer.getID(accounts[1])
0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63
>>> issuer.getInvestorRegistrar(id_)
0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8
```

6.3.5 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.

Setters

`IssuingEntity.`**`setCountry`**(*uint16 _country*, *bool _permitted*, *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
- `_permitted`: 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.

Emits the `CountryModified` event.

```
>>> issuer.setCountry(784, True, 1, [100, 0, 0, 0, 0, 0, 0, 0], {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x96f9a7e12e898fbd2fb6c7593a7ae82c4eea087c508929e616f86e98ae9b0db6
IssuingEntity.setCountry confirmed - block: 26   gas used: 116709 (1.46%)
<Transaction object
↪'0x96f9a7e12e898fbd2fb6c7593a7ae82c4eea087c508929e616f86e98ae9b0db6'>
```

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

Approve many countries at once.

- `_countries`: An array of country codes to modify
- `_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 or to explicitly restrict an entire country, use `IssuingEntity.setCountry`.

Emits the `CountryModified` event once for each country that is modified.

```
>>> issuer.setCountries([784],[1],[0], {'from': accounts[0]})

Transaction sent:␣
↪0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3
IssuingEntity.setCountries confirmed - block: 6   gas used: 72379 (0.90%)
<Transaction object
↪'0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3'>
```

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

Sets total investor limits, irrespective of country.

Emits the `InvestorLimitsSet` event.

```
>>> issuer.setInvestorLimits([2000, 500, 2000, 0, 0, 0, 0, 0], {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0xbeda494b5fb741ae659b866b9f5eca26b9add249ae75dc651a7944281e2ae4eb
IssuingEntity.setInvestorLimits confirmed - block: 27   gas used: 94926 (1.19%)
<Transaction object
↪'0xbeda494b5fb741ae659b866b9f5eca26b9add249ae75dc651a7944281e2ae4eb'>
```

Getters

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

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

```
>>> issuer.getInvestorCounts().dict()
{
    '_counts': ((1, 0, 1, 0, 0, 0, 0, 0),
    '_limits': (2000, 500, 2000, 0, 0, 0, 0, 0))
}
```

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

Returns the minimum rating, investor counts and investor limits for a given country. Countries that have not been set will return all zero values. The easiest way to verify if a country has been set is to check if `_minRating > 0`.

```
>>> issuer.getCountry(784).dict()
{
    '_count': (0, 0, 0, 0, 0, 0, 0, 0),
    '_limit': (100, 0, 0, 0, 0, 0, 0, 0),
    '_minRating': 1
}
```

6.3.6 Document Verification

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

Returns a recorded document hash. If no hash is recorded, it will return `0x00`.

See *Document Verification*.

```
>>> issuer.getDocumentHash("Shareholder Agreement")
"0xbeda494b5fb741ae659b866b9f5eca26b9add249ae75dc651a7944281e2ae4eb"
>>> issuer..getDocumentHash("Unknown Document")
0x0000000000000000000000000000000000000000000000000000000000000000
```

`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.

Emits the `NewDocumentHash` event.

```
>>> issuer.setDocumentHash("Shareholder Agreement",
↪"0xbeda494b5fb741ae659b866b9f5eca26b9add249ae75dc651a7944281e2ae4eb", {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3
IssuingEntity.setDocumentHash confirmed - block: 6   gas used: 72379 (0.90%)
<Transaction object
↪'0x7299b96013acb4661f4b7f05016c0de6726d2337032740aa29f5407cdabde0c3'>
```

6.3.7 Modules

Modules for token contracts are attached and detached through the associated `IssuingEntity`. This contract itself is not directly modular, however any module that declares it as the owner may be attached to all the associated token contracts.

See the *Modules* documentation for more information on module functionality and development.

Attaching and Detaching

`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.

```
>>> module = DividendModule.deploy(accounts[0], SecurityToken[0], issuer,␣
↪1600000000)

Transaction sent:␣
↪0x1b1e7a09e7731fcb724a6586e3cf71c07221db009e89445c33e07cc8e18e74d1
DividendModule.constructor confirmed - block: 13   gas used: 1756759 (21.96%)
DividendModule deployed at: 0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D
<DividendModule Contract object '0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D'>
>>>
>>> issuer.attachModule(SecurityToken[0], module, {'from': accounts[0]})

Transaction sent:␣
↪0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45
IssuingEntity.attachModule confirmed - block: 14   gas used: 212332 (2.65%)
<Transaction object
↪'0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45'>
```

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

Detaches a module.

```
>>> issuer.detachModule(SecurityToken[0], module, {'from': accounts[0]})

Transaction sent:␣
↪0xe1539492053b91ffb05dec6da6f73a02f0b3e44fcec707acf911d37922b65699
IssuingEntity.detachModule confirmed - block: 15   gas used: 28323 (0.35%)
<Transaction object
↪'0xe1539492053b91ffb05dec6da6f73a02f0b3e44fcec707acf911d37922b65699'>
```

6.3.8 Events

The `IssuingEntity` contract includes the following events.

`IssuingEntity.`**`TokenAdded`**(*address indexed token*)
Emitted after a new token contract has been associated via `IssuingEntity.addToken`.

`IssuingEntity.`**`RegistrarSet`**(*address indexed registrar*, *bool restricted*)
Emitted by `IssuingEntity.setRegistrar` when a new KYC registrar contract is added, or an existing registrar is restricted or permitted.

`IssuingEntity.`**`CustodianAdded`**(*address indexed custodian*)
Emitted when a new custodian contract is approved via `IssuingEntity.addCustodian`.

`IssuingEntity.`**`EntityRestriction`**(*bytes32 indexed id*, *bool restricted*)
Emitted whenever an investor or custodian has a restriction set or removed with `IssuingEntity.setEntityRestriction`.

`IssuingEntity.`**`TokenRestriction`**(*address indexed token*, *bool restricted*)
Emitted when a token restriction is set or removed via `IssuingEntity.setTokenRestriction`.

`IssuingEntity.`**`GlobalRestriction`**(*bool restricted*)
Emitted when a global restriction is set with `IssuingEntity.setGlobalRestriction`.

`IssuingEntity.`**`InvestorLimitsSet`**(*uint32[8] limits*)
Emitted when global investor limits are modified via `IssuingEntity.setInvestorLimits`.

`IssuingEntity.`**`CountryModified`**(*uint16 indexed country*, *bool permitted*, *uint8 minrating*, *uint32[8] limits*)
Emitted whenever country specific limits are set via `IssuingEntity.setCountry` or `IssuingEntity.SetCountries`.

`IssuingEntity.`**`NewDocumentHash`**(*string indexed document*, *bytes32 documentHash*)
Emitted when a new document hash is saved with `IssuingEntity.setDocumentHash`.

6.4 KYC

KYC registry contracts are whitelists that hold information on the identity, region, and rating of investors. Depending on the use case there are two implementations:

- KYCIssuer.sol is a streamlined whitelist contract designed for use with a single `IssuingEntity`.
- KYCRegistrar.sol is a more robust implementation. It is maintainable by one or more entities across many jurisdictions, and designed to supply KYC data to many `IssuingEntity` contracts.

Both contracts are derived from a common base KYC.sol that defines standard getter functions and events.

Contract authorities associate addresses to ID hashes that denote the identity of the investor who controls the address. More than one address may be associated to the same hash. Anyone can call `KYCBase.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.

6.4.1 Deployment

Deployment varies depending on the type of registrar contract.

KYCIssuer

The address of the issuer is declared during deployment. The identity of the contract owner and authorities are determined by calls to this contract.

`KYCIssuer.`**`constructor`**(*address _issuer*)

- `_issuer`: The address of the `IssuingEntity` contract to associate this contract with.

```
>>> issuer = IssuingEntity[0]
<IssuingEntity Contract object '0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06'>
>>> kyc = accounts[0].deploy(KYCIssuer, issuer)

Transaction sent:␣
↪0x98f595f75535df670d0c83b247bb471189938f0f57385fa1c7d3c6621748c703
KYCIssuer.constructor confirmed - block: 2   gas used: 1645437 (20.57%)
KYCIssuer deployed at: 0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8
<KYCIssuer Contract object '0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8'>
>>> kyc.issuer()
'0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06'
```

KYCRegistrar

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`.

```
>>> kyc = accounts[0].deploy(KYCRegistrar, [accounts[0]], 1)

Transaction sent:␣
↪0xd10264c1445aad4e9dc84e04615936624e0b96596fec2097bebc83f9d3e69664
KYCRegistrar.constructor confirmed - block: 2   gas used: 2853810 (35.67%)
KYCRegistrar deployed at: 0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06
<KYCRegistrar Contract object '0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06'>
```

6.4.2 Contract Owners and Authorities

KYCIssuer

`KYCIssuer` retrieves authority permissions and derives multisig functionality from the associated `IssuingEntity` contract. The owner of the issuer contract may call any method within the registrar. Authorities may call any method that they have been explicitely permitted to call.

See the *MultiSig Implementation* documentation for information on this aspect of the contract's functionality.

KYCRegistrar

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.

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. They 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.

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

Adding 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`

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

Emits the `NewAuthority` event.

```
>>> kyc.addAuthority([accounts[1], accounts[2]], [4, 11, 77, 784], 1, {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x6085f4c75f12c4bed01c541d9a7e1d8f7e1ffc85247b5582459cbdd99fa1b51b
KYCRegistrar.addAuthority confirmed - block: 2   gas used: 157356 (1.97%)
<Transaction object
↪'0x6085f4c75f12c4bed01c541d9a7e1d8f7e1ffc85247b5582459cbdd99fa1b51b'>
>>> id_ = kyc.getAuthorityID(accounts[1])
0x7b809759765e66e1999ae953ef432bec3472905be1588b398563de2912cd7d01
```

Modifying Authorities

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

Modifies the country permissions for an authority.

```
>>> kyc.isApprovedAuthority(accounts[1], 4)
True
>>> kyc.setAuthorityCountries(id_, [4, 11], False, {'from': accounts[0]})

Transaction sent:␣
↪0x60e9cc4c79bf08fd2929d33039f24278d63b28c91269ff79dc752f06a2c29e2a
KYCRegistrar.setAuthorityCountries confirmed - block: 3   gas used: 46196 (0.58%)
<Transaction object
↪'0x60e9cc4c79bf08fd2929d33039f24278d63b28c91269ff79dc752f06a2c29e2a'>
>>> kyc.isApprovedAuthority(accounts[1], 4)
False
```

`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.

```
>>> kyc.setAuthorityThreshold(id_, 2, {'from': accounts[0]})

Transaction sent:␣
↪0xe253c5acb5f0896ebdc92090c23bcec8baab0e23abe513ae6119caf51522e425
KYCRegistrar.setAuthorityThreshold confirmed - block: 4   gas used: 39535 (0.49%)
<Transaction object
↪'0xe253c5acb5f0896ebdc92090c23bcec8baab0e23abe513ae6119caf51522e425'>
>>>
>>> kyc.setAuthorityThreshold(id_, 3, {'from': accounts[0]})
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

KYCRegistrar.**setAuthorityRestriction**(*bytes32 _id*, *bool _restricted*)

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.

Emits the `AuthorityRestriction` event.

```
>>> kyc.isApprovedAuthority(accounts[1], 784)
True
>>> kyc.setAuthorityRestriction(id_, True)

Transaction sent:␣
↪0xeb3456fae407fb9bd67307536990376932 6c9f8699b313feb46e92f7f199c72e
KYCRegistrar.setAuthorityRestriction confirmed - block: 10   gas used: 40713 (28.
↪93%)
<Transaction object
↪'0xeb3456fae407fb9bd67307536990376932 6c9f8699b313feb46e92f7f199c72e'>
>>> kyc.isApprovedAuthority(accounts[1], 784)
False
```

Getters

The following getter methods are available to query information about contract authorities:

KYCRegistrar.**isApprovedAuthority**(*address _addr*, *uint16 _country*)

Checks whether an authority is approved to add or modify investors from a given country. Returns `false` if the authority is not permitted.

```
>>> kyc.isApprovedAuthority(accounts[1], 784)
True
```

KYCRegistrar.**getAuthorityID**(*address _addr*)

Given an address, returns the ID hash of the associated authority. If the address is not associated with an authority the call will revert.

```
>>> kyc.getAuthorityID(accounts[1])
0x7b809759765e66e1999ae953ef432bec3472905be1588b398563de2912cd7d01
>>> kyc.getAuthorityID(accounts[3])
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

6.4.3 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.

See the *Investor Data Standards* documentation for detailed information on how to generate and format investor information for use with registrar contracts.

Adding Investors

KYCBase.**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.

```
>>> id_ = kyc.generateID("JOHNDOE010119701234567890")
0xd3e7532ecb2c15babc9a5ac8e65f9d96b7030ab7e5dc9fffaa00ac15c0937be4
```

KYCBase.**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`.

Emits the `NewInvestor` event.

```
>>> kyc.addInvestor(id_, 784, "0x465500", 1, 9999999999, (accounts[3],), {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x47581e5b276298427f6a520353622b96cdecb29dff7269f03d7c957435398ebd
KYCRegistrar.addInvestor confirmed - block: 3   gas used: 120707 (1.51%)
<Transaction object
↪'0x47581e5b276298427f6a520353622b96cdecb29dff7269f03d7c957435398ebd'>
```

Modifying Investors

KYCBase.**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.

Emits the `UpdatedInvestor` event.

```
>>> kyc.updateInvestor(id_, "0x465500", 2, 1600000000, {'from': accounts[0]})

Transaction sent:␣
↪0xacfb17b530d2b565ea6016ab9b50051edb85e92e5ec6d2d85b1ac1708f897949
KYCRegistrar.updateInvestor confirmed - block: 4   gas used: 50443 (0.63%)
<Transaction object
↪'0xacfb17b530d2b565ea6016ab9b50051edb85e92e5ec6d2d85b1ac1708f897949'>
```

KYCBase.**setInvestorRestriction**(*bytes32 _id*, *bool _restricted*)

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

Emits the `InvestorRestriction` event.

```
>>> kyc.setInvestorRestriction(id_, True, {'from': accounts[0]})

Transaction sent:␣
↪0x175982346d2f00a25f00a69701cda6fa311d60ade94d801267f51eefa86dc49e
KYCRegistrar.setInvestorRestriction confirmed - block: 5   gas used: 41825 (0.52%)
<Transaction object
↪'0x175982346d2f00a25f00a69701cda6fa311d60ade94d801267f51eefa86dc49e'>
```

KYCRegistrar

The following method is only available in `KYCRegistrar`.

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.

```
>>> auth_id = kyc.getAuthorityID(accounts[1])
0x7b809759765e66e1999ae953ef432bec3472905be1588b398563de2912cd7d01
>>> kyc.setInvestorAuthority([id_], auth_id, {'from': accounts[0]})

Transaction sent:␣
↪0x175982346d2f00a25f00a69701cda6fa311d60ade94d801267f51eefa86dc49e
KYCRegistrar.setInvestorRestriction confirmed - block: 5   gas used: 41825 (0.52%)
<Transaction object
↪'0x175982346d2f00a25f00a69701cda6fa311d60ade94d801267f51eefa86dc49e'>
```

6.4.4 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.

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

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

In `KYCRegistrar`: 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.

Emits the `RegisteredAddresses` event.

```
>>> kyc.registerAddresses(id_, [accounts[4], accounts[5]], {'from': accounts[0]})

Transaction sent:␣
↪0xf508d5c72a1f707d88a0af4dbfc1007ecf2a7f04aa53bfcba2862e46fe3e647d
KYCRegistrar.registerAddresses confirmed - block: 7   gas used: 60329 (0.75%)
<Transaction object
↪'0xf508d5c72a1f707d88a0af4dbfc1007ecf2a7f04aa53bfcba2862e46fe3e647d'>
```

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

Restricts one or more addresses associated with an ID.

In `KYCRegistrar`: 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.

Emits the `RestrictedAddresses` event.

```
>>> kyc.restrictAddresses(id_, [accounts[4]], {'from': accounts[0]})

Transaction sent:␣
↪0xfeb1b2316b3c35b2e08d84b3922030b97e671eec799d0fb0eaf748f69ab0866b
KYCRegistrar.restrictAddresses confirmed - block: 8   gas used: 60533 (0.76%)
<Transaction object
↪'0xfeb1b2316b3c35b2e08d84b3922030b97e671eec799d0fb0eaf748f69ab0866b'>
```

6.4.5 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.

Calls that Return False

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

`KYCBase`.**`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.

```
>>> kyc.getID(accounts[1])
0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a
>>> kyc.getID(accounts[2])
0x0000000000000000000000000000000000000000000000000000000000000000
```

KYCBase.**isRegistered**(*bytes32 _id*)

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

```
>>> kyc.isRegistered(
↪'0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a')
True
>>> kyc.isRegistered(
↪'0x81a5c449c2409c87d702e0c4a675313347faf1c39576af357dd75efe7cad4793')
False
```

KYCBase.**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?

```
>>> kyc.isPermitted(accounts[1])
True
>>> kyc.isPermitted(accounts[2])
False
```

KYCBase.**isPermittedID**(*bytes32 _id*)

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

```
>>> kyc.isPermittedID(
↪'0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a(')
True
>>> kyc.isPermittedID(
↪'0x81a5c449c2409c87d702e0c4a675313347faf1c39576af357dd75efe7cad4793')
False
```

Calls that Revert

The remaining calls will revert under some conditions:

KYCBase.**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.

```
>>> kyc.getInvestor(a[1]).dict()
{
    '_country': 784,
    '_id': "0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a",
    '_permitted': True,
    '_rating': 1
}
>>> kyc..getInvestor(a[0])
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Address␣
↪not registered
```

KYCBase.**getInvestors**(*address _from*, *address _to*)

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

```
>>> kyc.getInvestors(accounts[1], accounts[2]).dict()
{
    '_country': (784, 784),
    '_id': ("0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a",
↪"0x9becd445b3c5703a4f1abc15870dd10c56bb4b4a70c68dba05e116551ab11b44"),
    '_permitted': (True, False),
    '_rating': (1, 2)
}
>>> kyc.getInvestors(accounts[1], accounts[3])
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert Receiver␣
↪not Registered
```

KYCBase.**getRating**(*bytes32 _id*)

Returns the investor rating number for a given ID.

Reverts if the ID is not registered.

```
>>> kyc.getRating(
↪"0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a")
1
>>> kyc.getRating("0x00")
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

KYCBase.**getRegion**(*bytes32 _id*)

Returns the investor region code for a given ID.

Reverts if the ID is not registered.

```
>>> kyc.getRegion(
↪"0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a")
0x653500
>>> kyc.getRegion("0x00")
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

KYCBase.**getCountry**(*bytes32 _id*)

Returns the investor country code for a given ID.

Reverts if the ID is not registered.

```
>>> kyc.getCountry(
↪"0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a")
784
>>> kyc.getCountry("0x00")
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

KYCBase.**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.

```
>>> kyc.getExpires(
↪"0x1d285a37d3afce3a200a1eeb6697e59d15e8dc0d9b5132391e3ee53c7a69f18a")
1600000000
>>> kyc.getExpires("0x00")
File "contract.py", line 277, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

6.4.6 Events

Both KYC implementations include the following events.

The `authority` value in each event is the ID hash of the authority that called the method where the event was emitted.

KYCBase.**NewInvestor**(*bytes32 indexed id*, *uint16 indexed country*, *bytes3 region*, *uint8 rating*, *uint40 expires*, *bytes32 indexed authority*)

Emitted when a new investor is added to the registry with `KYCBase.addInvestor`.

KYCBase.**UpdatedInvestor**(*bytes32 indexed id*, *bytes3 region*, *uint8 rating*, *uint40 expires*, *bytes32 indexed authority*)

Emitted when data about an existing investor is modified with `KYCBase.updateInvestor`.

KYCBase.**InvestorRestriction**(*bytes32 indexed id*, *bool permitted*, *bytes32 indexed authority*)

Emitted when a restriction upon an investor is set or removed with `KYCBase.setInvestorRestriction`.

KYCBase.**RegisteredAddresses**(*bytes32 indexed id*, *address[] addr*, *bytes32 indexed authority*)

Emitted by `KYCBase.registerAddresses` when new addresses are associated with an investor ID, or existing addresses have a restriction removed.

KYCBase.**RestrictedAddresses**(*bytes32 indexed id*, *address[] addr*, *bytes32 indexed authority*)

Emitted when a restriction is set upon addresses associated with an investor ID with `KYCBase.restrictAddresses`.

KYCRegistrar

The following events are specific to `KYCRegistrar`'s authorities:

KYCRegistrar.**NewAuthority**(*bytes32 indexed id*)

Emitted when a new authority is added via `KYCRegistrar.addAuthority`.

KYCRegistrar.**AuthorityRestriction**(*bytes32 indexed id*, *bool permitted*)

Emitted when an authority is restricted or permitted via `KYCRegistrar.setAuthorityRestriction`.

6.5 Custodians

Custodian contracts are approved 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 section:

- OwnedCustodian.sol: Standard owned custodian contract with `Multisig` and `Modular` functionality.
- IBaseCustodian.sol: The minimum contract interface required for a custodian to interact with an `IssuingEntity` contract.

Warning: An issuer should not approve a Custodian if the contract source code cannot be verified, or it is using a non-standard implementation that has not undergone a thorough audit. The SFT protocol includes a standard owned Custodian contract that allows for customization through modules.

6.5.1 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 balances are tracked directly in the corresponding token contract and can be queried through `TokenBase.custodianBalanceOf`.

```
>>> cust
<OwnedCustodian Contract object '0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D'>
>>> token.balanceOf(accounts[1])
10000
>>> token.custodianBalanceOf(accounts[1], cust)
0
>>> token.balanceOf(cust)
0
>>> token.transfer(cust, 5000, {'from': accounts[1]})

Transaction sent: 0x4b09b29216d130dc06798ee673759a4e77e4823655c6477e895242f027726412
SecurityToken.transfer confirmed - block: 16   gas used: 155761 (1.95%)
<Transaction object
↪'0x4b09b29216d130dc06798ee673759a4e77e4823655c6477e895242f027726412'>
>>> token.balanceOf(accounts[1])
5000
>>> token.custodianBalanceOf(accounts[1], cust)
5000
```

(continues on next page)

(continued from previous page)

```
>>> token.balanceOf(cust)
5000
```

Token Transfers

There are three types of token transfers related to Custodians.

- **Inbound**: transfers from an investor into the Custodian contract.
- **Outbound**: transfers out of the Custodian contract to an investor's wallet.
- **Internal**: transfers involving a change of ownership within the Custodian contract. This is the only type of transfer that involves a change of ownership of the token, however no tokens actually move.

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.

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 token methods relating to internal transfers.

`TokenBase.`**`checkTransferCustodian`**(*address _cust*, *address _from*, *address _to*, *uint256 _value*)
Checks if a custodian internal transfer of tokens is permitted.

- `_cust`: Address of the custodian
- `_from`: Address of the sender
- `_to`: Address of the recipient
- `_value`: Amount of tokens to be transferred

Returns `true` if the transfer is permitted. If the transfer is not permitted, the call will revert with the reason given in the error string.

Permissioning checks for custodial transfers are identical to those of normal transfers.

`SecurityToken.`**`transferCustodian`**(*address[2] _addr*, *uint256 _value*)
Modifies investor counts and ownership records based on an internal transfer of ownership within the Custodian contract.

- `_addr`: Array of sender and receiver addresses.
- `_value`: Amount of tokens being transferred

Minimal Implementation

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

`IBaseCustodian.`**`ownerID`**`()`
Public bytes32 hash representing the owner of the contract.

`IBaseCustodian.`**`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.

6.5.2 OwnedCustodian

`OwnedCustodian` is a standard custodian implementation that is controlled and maintained by a known legal entity. Use cases for this contract may 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.

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

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 `OwnedCustodian.ownerID`.

Once deployed, the custodian must be approved by an `IssuingEntity` before it can receive tokens associated with that contract.

```
>>> cust = accounts[0].deploy(OwnedCustodian, [accounts[0]], 1)

Transaction sent:␣
↪0x11540767a467504e3ddd03c8c2423840a69bd82a6f28db33ea869570b87486f0
OwnedCustodian.constructor confirmed - block: 13   gas used: 3326386 (41.58%)
OwnedCustodian deployed at: 0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D
<OwnedCustodian Contract object '0x3BcC6Ad6CFbB1997eb9DA056946FC38a6b5E270D'>
```

Public Constants

The following public variables cannot be changed after contract deployment.

`OwnedCustodian.`**`ownerID`**`()`
The bytes32 ID hash of the contract owner.

```
>>> cust.ownerID()
0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63
```

Balances and Transfers

Checking Balances

Custodied investor balances are tracked within the token contract. They can be queried using `TokenBase.custodianBalanceOf` or `OwnedCustodian.balanceOf`.

`OwnedCustodian.`**`balanceOf`**`(`*address _token*, *address _owner*`)`
Returns the custodied token balance for a given investor address.

```
>>> cust.balanceOf(token, accounts[1])
5000
>>> token.custodianBalanceOf(accounts[1], cust)
5000
```

Checking Transfer Permissions

`OwnedCustodian.`**`checkCustodianTransfer`**`(`*address _token*, *address _from*, *address _to*, *uint256 _value*`)`
Checks if an internal transfer is permitted.

- `_token`: Token address
- `_from`: Sender address
- `_to`: Receiver address

- `_value`: Amount to transfer

Returns `true` if the transfer is permitted. If it is not, the call will revert with the reason given in the error string.

Permissioning checks for custodial transfers are identical to those of normal transfers.

```
>>> cust.balanceOf(token, accounts[1])
2000
>>> cust.checkCustodianTransfer(token, accounts[1], accounts[2], 1000)
True
>>> cust.checkCustodianTransfer(token, accounts[1], accounts[2], 5000)
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert␣
↪Insufficient Custodial Balance
```

Transferring Tokens

`OwnedCustodian.`**`transferInternal`**(*address _token*, *address _from*, *address _to*, *uint256 _value*)

- `_token`: SecurityToken address
- `_from`: Sender address
- `_to`: Receiver address
- `_value`: Amount to transfer

```
>>> cust.transferInternal(token, accounts[1], accounts[2], 5000, {'from':␣
↪accounts[0]})

Transaction sent:␣
↪0x1c5cf1d01d2d5f9b9d9e801d8e2a0b9b2eb50fa11fbe03864b69ccf0fe2c03fc
OwnedCustodian.transferInternal confirmed - block: 17   gas used: 189610 (2.37%)
<Transaction object
↪'0x1c5cf1d01d2d5f9b9d9e801d8e2a0b9b2eb50fa11fbe03864b69ccf0fe2c03fc'>
```

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

Transfers tokens out of the Custodian contract.

- `_token`: Token address
- `_to`: Receipient address
- `_value`: Amount to transfer

```
>>> cust.transfer(token, accounts[2], 5000, {'from': accounts[0]})

Transaction sent:␣
↪0x227f7c24d68d63aa567c16458e039a283481ef5fd79d8b9e48c88b033ff18f79
OwnedCustodian.transfer confirmed - block: 18   gas used: 149638 (1.87%)
<Transaction object
↪'0x227f7c24d68d63aa567c16458e039a283481ef5fd79d8b9e48c88b033ff18f79'>
```

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. Only callable by the owner or an approved authority.

```
>>> cust.attachModule(module, {'from': accounts[0]})

Transaction sent:␣
↪0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45
OwnedCustodian.attachModule confirmed - block: 14   gas used: 212332 (2.65%)
<Transaction object
↪'0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45'>
```

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

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

```
>>> cust.detachModule(module, {'from': accounts[0]})

Transaction sent:␣
↪0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45
OwnedCustodian.detachhModule confirmed - block: 15   gas used: 43828 (2.65%)
<Transaction object
↪'0x7123091c968dbe0c279aa6850c668534aef327972a08d65b67779108cbaa9b45'>
```

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

Returns `true` if a module is currently active on the contract, `false` if not.

```
>>> cust.isActiveModule(cust_module)
True
>>> cust.isActiveModule(other_module)
False
```

`Modular.`**`isPermittedModule`**(*address _module*, *bytes4 _sig*)

Returns `true` if a module is active on the contract, and permitted to call the given method signature. Returns `false` if not permitted.

```
>>> cust.isPermittedModule(cust_module, "0x40c10f19")
True
>>> cust.isPermittedModule(cust_module, "0xc39f42ed")
False
```

Events

`OwnedCustodian` includes the following events:

`OwnedCustodian.`**`ReceivedTokens`**(*address indexed token*, *address indexed from*, *uint256 amount*)

Emitted by `OwnedCustodian.receiveTransfer` when tokens are sent into the custodian contract.

`OwnedCustodian.`**`SentTokens`**(*address indexed token*, *address indexed to*, *uint256 amount*)

Emitted by `OwnedCustodian.transfer` after tokens are sent out of the custodian contract.

`OwnedCustodian.`**`TransferOwnership`**(*address indexed token*, *address indexed from*, *address indexed to*, *uint256 value*)

Emitted by `OwnedCustodia.transferInternal` after an internal change of beneficial ownership.

6.6 MultiSig Implementation

IssuingEntity and *Custodians* 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. See the *KYCRegistrar* documentation for more information.

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

Note: In the code examples `MultiSig` is deployed as a standalone contract for simplicity. In a production environment it should be included as an inherited contract, not deployed directly.

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

```
>>> ms = accounts[0].deploy(MultiSig, [accounts[0], accounts[1]], 1)
Transaction sent:␣
 ↪0xba276d522a5d7f99670df3640053deabb0f97b4e545be0922aeedb48b4af98cf
MultiSig.constructor confirmed - block: 1   gas used: 1875646 (23.45%)
MultiSig deployed at: 0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8
<MultiSig Contract object '0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8'>
```

6.6.2 Public Constants

The following public variables cannot be changed after contract deployment.

`MultiSig.`**`ownerID`**`()`

The bytes32 ID hash of the issuer.

```
>>> ms.ownerID()
0xce1e12589ad8fb3eed11af5b9ef8788c25b574d4073d23c871e003021400c429
```

6.6.3 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.

Setters

`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.

The ID of the authority is generated from a keccak of the initial addresses associated with the authority.

Emits the `NewAuthority`, `NewAuthorityAddresses` and `NewAuthorityPermissions` events.

```
>>> ms.addAuthority([accounts[2], accounts[3]], ["0xfb6e54f9", "0xd0370e78"],␣
↪2000000000, 1, {'from': accounts[0]})

Transaction sent:␣
↪0xc3a7aa469048e288030aa8eaf90f8e12b369695ad4f487ac43efe05add1a042b
MultiSig.addAuthority confirmed - block: 2   gas used: 160697 (2.01%)
<Transaction object
↪'0xc3a7aa469048e288030aa8eaf90f8e12b369695ad4f487ac43efe05add1a042b'>
>>>
>>> id_ = ms.getID(accounts[2])
0x857bfe5ad6c226322d3b517d158f60ac64e53b7b500d1ac2f27117cdf911a9c6
```

`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.

Emits the `ApprovedUntilSet` event.

```
>>> ms.setAuthorityApprovedUntil(id_, 3000000000, {'from': accounts[0]})

Transaction sent:␣
↪0x321652c5d0cdb2d8dd6b6e6123bc8e48bdf5a745378dabf2bb3ff5944f5a9ba9
MultiSig.setAuthorityApprovedUntil confirmed - block: 3   gas used: 42055 (0.53%)
<Transaction object
↪'0x321652c5d0cdb2d8dd6b6e6123bc8e48bdf5a745378dabf2bb3ff5944f5a9ba9'>
```

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

Modifies call permissions for an authority.

> **Warning:** If an external contract method using `checkMultiSigExternal` has the same signature as one inside the multi-sig contract, it will be impossible to set unique permissions for each function. Developers and auditors of external contracts should always keep this in mind.

If permission is granted, emits the `NewAuthorityPermissions` event. If permission is revoked, emits the `RemovedAuthorityPermissions` event.

```
>>> ms.setAuthoritySignatures(id_, ["0xfb6e54f9"], False, {'from': accounts[0]})

Transaction sent:␣
↪0x5381f0d788c5fcf9db82a9c36696648d2cd0bfbf77dcfed99169102f37999622
MultiSig.setAuthoritySignatures confirmed - block: 4   gas used: 28392 (0.35%)
<Transaction object
↪'0x5381f0d788c5fcf9db82a9c36696648d2cd0bfbf77dcfed99169102f37999622'>
```

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

Modifies the multisig threshold requirement for an authority. The owner may call to modify the threshold for any authority. An authority that has been permitted to call this function may call to modify their own threshold.

Emits the `ThresholdSet` event.

```
>>> ms.setAuthorityThreshold(id_, 1, {'from': accounts[0]})

Transaction sent:␣
↪0x37856411734aa9e354a265a73f143a66efadf4c7a3c94078817b430c0108d261
MultiSig.setAuthorityThreshold confirmed - block: 9   gas used: 41376 (29.27%)
<Transaction object
↪'0x37856411734aa9e354a265a73f143a66efadf4c7a3c94078817b430c0108d261'>
>>> ms.setAuthorityThreshold.call(id_, 3, {'from': accounts[0]})
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert dev:␣
↪threshold too high
```

`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.

Emits the `NewAuthorityAddresses` event.

```
>>> ms.addAuthorityAddresses(id_, [accounts[4]], {'from': accounts[0]})

Transaction sent:␣
↪0xe7654ccaaa1f9c70c958bffae9c3ce8c58289b446a83d1e746ddac090ef830c6
MultiSig.addAuthorityAddresses confirmed - block: 10   gas used: 66482 (39.93%)
<Transaction object
↪'0xe7654ccaaa1f9c70c958bffae9c3ce8c58289b446a83d1e746ddac090ef830c6'>
```

`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.

Emits the `RemovedAuthorityAddresses` event.

```
>>> ms.removeAuthorityAddresses(id_, [accounts[4]], {'from': accounts[0]})

Transaction sent:␣
↪0x020d9f20ddafe91490276527ac1d4c55965ec6137dd8513025838029ab1af39b
MultiSig.removeAuthorityAddresses confirmed - block: 11   gas used: 65962 (39.75%)
<Transaction object
↪'0x020d9f20ddafe91490276527ac1d4c55965ec6137dd8513025838029ab1af39b'>
```

Getters

There are several getter methods available for to query information about multisig authorities. In some cases these calls will revert if no data is found.

Calls that Return False

`MultiSig.`**`isAuthority`**(*address _addr*)

Checks if an address is associated with an authority.

```
>>> ms.isAuthority(accounts[3])
True
>>> ms.isAuthority(accounts[5])
False
```

`MultiSig.`**`isAuthorityID`**(*bytes32 _id*)

Checks if an ID hash is one belonging to an authority.

```
>>> ms.isAuthorityID(id_)
True
>>> ms.isAuthorityID("0x1234")
False
```

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

Returns the authority ID associted with a given address. If the address is not associated with an authority, returns `0x00`.

```
>>> id_ = ms.getID(accounts[2])
0x857bfe5ad6c226322d3b517d158f60ac64e53b7b500d1ac2f27117cdf911a9c6
>>> ms.getID(accounts[6])
0x0000000000000000000000000000000000000000000000000000000000000000
```

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

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

This call is only a general check to see if the authority may call to the method. Specific logic within any given method may still prevent this authority from completing the call.

```
>>> ms.isApprovedAuthority(accounts[3], "0xd0370e78")
True
>>> ms.isApprovedAuthority(accounts[5], "0xd0370e78")
False
>>> ms.isApprovedAuthority(accounts[3], "0x932324e5")
False
```

Calls that Revert

The remaining calls will revert under some conditions:

`MultiSig.`**`getAuthority`**(*bytes32 _id*)
Given an authority ID, returns the number of approved addresses, epoch time the authority is approved until, and multisig threshold value.

If the ID is not associated with an authority the call will revert.

```
>>> ms.getAuthority(id_).dict()
{
    '_addressCount': 2,
    '_approvedUntil': 3000000000,
    '_threshold': 1
}
>>> ms.getAuthority('0x1234')
File "contract.py", line 282, in call
  raise VirtualMachineError(e)
VirtualMachineError: VM Exception while processing transaction: revert
```

6.6.4 Implementing in other Contracts

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`**(*address _caller*, *bytes32 _callHash*, *bytes4 _sig*)
External function, used to implement multisig in an external contract.

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

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

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

6.6.5 Events

`MultiSig` includes the following events.

`MultiSig.`**`MultiSigCall`**(*bytes32 indexed id*, *bytes4 indexed callSignature*, *bytes32 indexed callHash*, *address caller*, *uint256 callCount*, *uint256 threshold*)
Emitted whenever a multisig call is made, but the threshold has not been reached.

`MultiSig.`**`MultiSigCallApproved`**(*bytes32 indexed id*, *bytes4 indexed callSignature*, *bytes32 indexed callHash*, *address caller*)
Emitted when a multisig call is made, and the required threshold to complete the call is reached.

`MultiSig.`**`NewAuthority`**(*bytes32 indexed id*, *uint32 approvedUntil*, *uint32 threshold*)
Emitted when a new multisig authority is added via `MultiSig.addAuthority`.

`MultiSig.`**`NewAuthorityAddresses`**(*bytes32 indexed id*, *address[] added*, *uint32 ownerCount*)
Emitted when new addresses are associted to an authority, either via `MultiSig.addAuthority` or `MultiSig.addAuthorityAddresses`.

`MultiSig.`**`RemovedAuthorityAddresses`**(*bytes32 indexed id*, *address[] removed*, *uint32 ownerCount*)
Emitted when authority addresses are removed via `MultiSig.removeAuthorityAddresses`.

`MultiSig.`**`ApprovedUntilSet`**(*bytes32 indexed id*, *uint32 approvedUntil*)
Emitted when an authority's approval time is modified via `MultiSig.setAuthorityApprovedUntil`.

`MultiSig.`**`ThresholdSet`**(*bytes32 indexed id*, *uint32 threshold*)
Emitted when an authority's threshold is modified via `MultiSig.setAuthorityThreshold`.

`MultiSig.`**`NewAuthorityPermissions`**(*bytes32 indexed id*, *bytes4[] signatures*)
Emitted when an authority is given new method permissions, either via `MultiSig.addAuthority` or `MultiSig.setAuthoritySignatures`.

`MultiSig.`**`RemovedAuthorityPermissions`**(*bytes32 indexed id*, *bytes4[] signatures*)
Emitted when an authority has a method permission removed via `MultiSig.setAuthoritySignatures`.

6.7 Modules

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

Modules introduce functionality in two ways:

- **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.
- **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.

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

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.
- Module.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 auditted, ensure you understand exactly what it does, and be **very** wary of any module that requires permissions outside of it's documented behaviour.

6.7.1 Attaching and Detaching

Modules are attached or detached via the `attachModule` and `detachModule` methods. For *Custodians* modules this method is available within `OwnedCustodian`, for token modules it is called via the associated `IssuingEntity` contract.

Ownership

Each module has an owner which is typically set during deployment. If the owner is set as an `IssuingEntity` contract, it may be attached to every token associated with that issuer. In this way a single module can add functionality or permissioning to many tokens.

`ModuleBase.`**`getOwner`**`()`
Returns the address of the parent contract that the module is owned by.

Declaring Hooks and Permissions

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

`ModuleBase.`**`getPermissions`**`()`
Returns the following:

- `permissions`: `bytes4` array of method signatures within the parent contract that the module is permitted to call.
- `hooks`: `bytes4` array of method signatures within the module that the parent contract may call into.
- `hookBools`: A `uint256` bit field. The first 128 bits set if each hook is active initially, the second half sets if each hook should be always called. See *Bit Fields*.

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.

Bit Fields

Solidity uses 8 bits to store a boolean, however only 1 bit is required. To maximize gas efficiency when handling many booleans at once, `ModuleBase` uses bit fields.

This functionality is mostly handled within internal methods and not user-facing, with one notable exception: in `ModuleBase.getPermissions` the return value `hookBools` is a set of 2 x 128 byte bit fields given as a single `uint256`. The first determines if each hook is active initially, the second if the hook method should always be called when it is active.

Each bit field is read **right to left**.

For example: suppose you have declared four hook points for a module. The first, third and fourth should be active initially. The third should always called when active.

Written left to right, the two bit fields will look like:

```
active = "1011"
always = "0010"
```

Now we must reverse them, and left-pad with zeros to 128 bits:

```
>>> active = active[::-1].zfill(128)
>>> print(active)
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>>
>>> always = always[::-1].zfill(128)
>>> print(always)
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Finally we join the two strings, and then convert the value from binary to decimal:

```
>>> final = active + always
>>> int(final, 2)
44236707699722000250238698966129867489 32
```

The following python function can be used to quickly convert two boolean lists into a uint256 to be used as `hookBools`:

```
def generate_bitfield(active: list, always: list) -> int:
    assert len(active) == len(always)
    assert len(active) <= 128
    active = "".join("1" if i else "0" for i in active[::-1]).zfill(128)
    always = "".join("1" if i else "0" for i in always[::-1]).zfill(128)
    return int(active + always, 2)
```

Note: If all your hooks are to be active initially and always called, the simplest approach is to set `hookBool = uint256(-1)`. Underflowing uint256 in this way results in 256 bits set to 1.

Checking Active Modules

The following getter is available in the parent contract, to check if a module is currently active:

`Modular.`**`isActiveModule`**`(`*address _module*`)`
Returns `true` if a module is currently active on the contract.

6.7.2 Permissioning

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.

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

Checking Permissions

Any call from a module to a function within the parent contract must first pass a check by this method:

`Modular.`**`isPermittedModule`**(*address _module*, *bytes4 _sig*)

Returns `true` if a module is active on the contract, and permitted to call the given method signature. Returns `false` if not permitted.

Callable Parent Methods

Modules may be permitted to call the following parent methods:

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

SecurityToken

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

- Permission signature: `0x23b872dd`

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

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

`TokenBase.`**`modifyAuthorizedSupply`**(*uint256 _value*)

- Permission signature: `0xc39f42ed`

Modifies the authorized supply.

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

- Permission signature: `0x40c10f19`

Mints new tokens to the given address.

Calling this method will also call any hooked in `STModule.totalSupplyChanged` and `IssuerModule.tokenTotalSupplyChanged` methods.

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

- Permission signature: `0x9dc29fac`

Burns tokens at the given address.

Calling this method will also call any hooked in `STModule.totalSupplyChanged` and `IssuerModule.tokenTotalSupplyChanged` methods.

`TokenBase.`**`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.

NFToken

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

- Permission signature: `0x23b872dd`

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

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

`TokenBase.`**`modifyAuthorizedSupply`**(*uint256 _value*)

- Permission signature: `0xc39f42ed`

Modifies the authorized supply.

`NFToken.`**`mint`**(*address _owner*, *uint48 _value*, *uint32 _time*, *bytes2 _tag*)

- Permission signature: `0x15077ec8`

Mints new tokens to the given address.

Calling this method will also call any hooked in `NFTModule.totalSupplyChanged` and `IssuerModule.tokenTotalSupplyChanged` methods.

`NFToken.`**`burn`**(*uint48 _start*, *uint48 _stop*)

- Permission signature: `0x9a0d378b`

Burns tokens at the given address.

Calling this method will also call any hooked in `NFTModule.totalSupplyChanged` and `IssuerModule.tokenTotalSupplyChanged` methods.

`NFToken.`**`modifyRange`**(*uint48 _pointer*, *uint32 _time*, *bytes2 _tag*)

- Permission signature: `0x712a516a`

Modifies the time restriction and tag for a single range.

`NFToken.`**`modifyRanges`**(*uint48 _start*, *uint48 _stop*, *uint32 _time*, *bytes2 _tag*)

- Permission signature: `0x786500aa`

Modifies the time restriction and tag for all tokens within a given range.

`TokenBase.`**`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.

Custodian

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

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

- Permission signature: `0xbeabacc8`

Transfers tokens from the custodian to an investor.

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

`OwnedCustodian.`**`transferInternal`**(*address _token*, *address _from*, *address _to*, *uint256 _value*)

- Permission signature: `0x2f98a4c3`

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

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

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

- Permission signature: `0xbb2a8522`

Detaches a module.

6.7.3 Hooks and Tags

- **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.
- **Tags** are `bytes2` values attached to token ranges in `NFToken`, that allow for more granular hook attachments.

Hooks and tags are defined in the following struct:

```
struct Hook {
    uint256[256] tagBools;
    bool permitted;
    bool active;
    bool always;
}
```

- `tagBools`: An bit field of length `2^16`. Defines granular hook points based on specific tags.
- `permitted`: Can only be set the first time the module is attached. If `true`, this is an available hook point for the module.
- `active`: Set during attachment, can be modified by the module. If `true`, this hook is currently active and will be called during the execution of the parent module.
- `always`: Set during attachment, can be modified by the module. If `true`, this hook is always called regardless of the tag value.

Hooks involving tokens from an `NFToken` contract rely upon tags to determine if the hook point should be called. A tag is a `bytes2` that is assigned to a specific range of tokens. When a hook point involves a tagged token range, the following three conditions are evaluated to see if the hook method should be called:

- Is `Hook.always` set to `true`?
- Is the first byte of the tag, followed by '00', set to true within `Hook.tagBools`?
- Is the entire tag set to true within `Hook.tagBools`?

For example, if the tag is `0xff32`, the hook point will be called if either `Hook.always`, `Hook.tagBools[0xff00]`, or `Hook.tagBools[0xff32]` are `true`.

For hook points that do not involve tags, the module should set `active` and `always` to true when it wishes to be called.

Setting and Modifying

Modules can be designed to modify their own active hook points and tag settings as they progress through different stages of functionality. Avoiding unnecessary external calls from hook points to modules that are no longer relevent helps keep gas costs down.

The following methods are used to modify hook and tag settings. These methods may only be called from the module while it is active.

`Modular.`**`setHook`**(*bytes4 _sig*, *bool _active*, *bool _always*)
Enables or disables a hook point for an active module.

- `_sig`: Signature of the hooked method.
- `_active`: Boolean for if hooked method is active.
- `_always`: Boolean for if hooked method should always be called when active.

`Modular.`**`setHookTags`**(*bytes4 _sig*, *bool _value*, *bytes1 _tagBase*, *bytes1[] _tags*)
Enables or disables specific tags for a hook point.

- `_sig`: Signature of the hooked method.
- `_value`: Boolean value to set each tag to.
- `_tagBase`: The first byte of the tag to set.
- `_tags`: Array of 2nd bytes for the tag.

For example: if `_tagBase = 0xff` and `_tags = [0x11, 0x22]`, you will modify tags `0xff00`, `0xff11`, and `0xff22`.

`Modular.`**`clearHookTags`**(*bytes4 _sig*, *bytes1[] _tagBase*)
Disables many tags for a given hook point.

- `_sig`: Signature of the hooked method.
- `_tagBase`: Array of first bytes for tags to disable.

For example: if `_tagBase = [0xee, 0xff]` it will clear tags `0xee00`, `0xee01 ... 0xeeff`, and `0xff00`, `0xff01 ... 0xffff`.

Hookable Module Methods

The following methods may be included in modules and given as hook points via `getPermissions`.

Inputs and outputs of all hook points are also defined in IModules.sol. This can be a useful starting point when writing your own modules.

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: `0x8b5f1240`

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

- Hook signature: `0x741b5078`

Called after the total supply has been modified by `SecurityToken.mint` or `SecurityToken.burn`.

- `_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.

NFToken

`NFToken` contracts also include all the hook points for `SecurityToken`.

Hook points that are unique to `NFToken` also perform a check against the tag of the related range before calling to a module.

`NFTModule.`**`checkTransferRange`**(*address[2] _addr*, *bytes32 _authID*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint48[2] _range*)

- Hook signature: `0x2d79c6d7`

Called by `NFToken.checkTransfer` and `NFToken.transferRange` to verify if the transfer of a specific range 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.
- `_range`: Start and stop index of token range.

`NFTModule.`**`transferTokenRange`**(*address[2] _addr*, *bytes32[2] _id*, *uint8[2] _rating*, *uint16[2] _country*, *uint48[2] _range*)

- Hook signature: `0xead529f5`

Called after a token range has been transferred successfully with `NFToken.transfer`, ``NFToken.transferFrom` or `NFToken.transferRange`.

- `_addr`: Sender and receiver addresses.
- `_id`: Sender and receiver IDs.
- `_rating`: Sender and receiver investor ratings.
- `_country`: Sender and receiver countriy codes.
- `_range`: Start and stop index of token range.

Custodian

`CustodianModule.`**`sentTokens`**(*address _token*, *address _to*, *uint256 _value*)

- Hook signature: `0xb4684410`

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

- `_token`: Address of token that was sent.
- `_to`: Address of the recipient.
- `_value`: Number of tokens that were sent.

`CustodianModule.`**`receivedTokens`**(*address _token*, *address _from*, *uint256 _value*)

- Hook signature: `0xb15bcbc4`

Called after a tokens have been transferred into a Custodian.

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

`CustodianModule.`**`internalTransfer`**(*address _token*, *address _from*, *address _to*, *uint256 _value*)

- Hook signature: `0x44a29e2a`

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

- `_token`: Address of token that was received.
- `_from`: Address of the sender.

- `_to`: Address of the recipient.
- `_value`: Number of tokens that were received.

6.7.4 Module Execution Flows

The following diagrams show the sequence in which modules are called during some of the more complex methods.

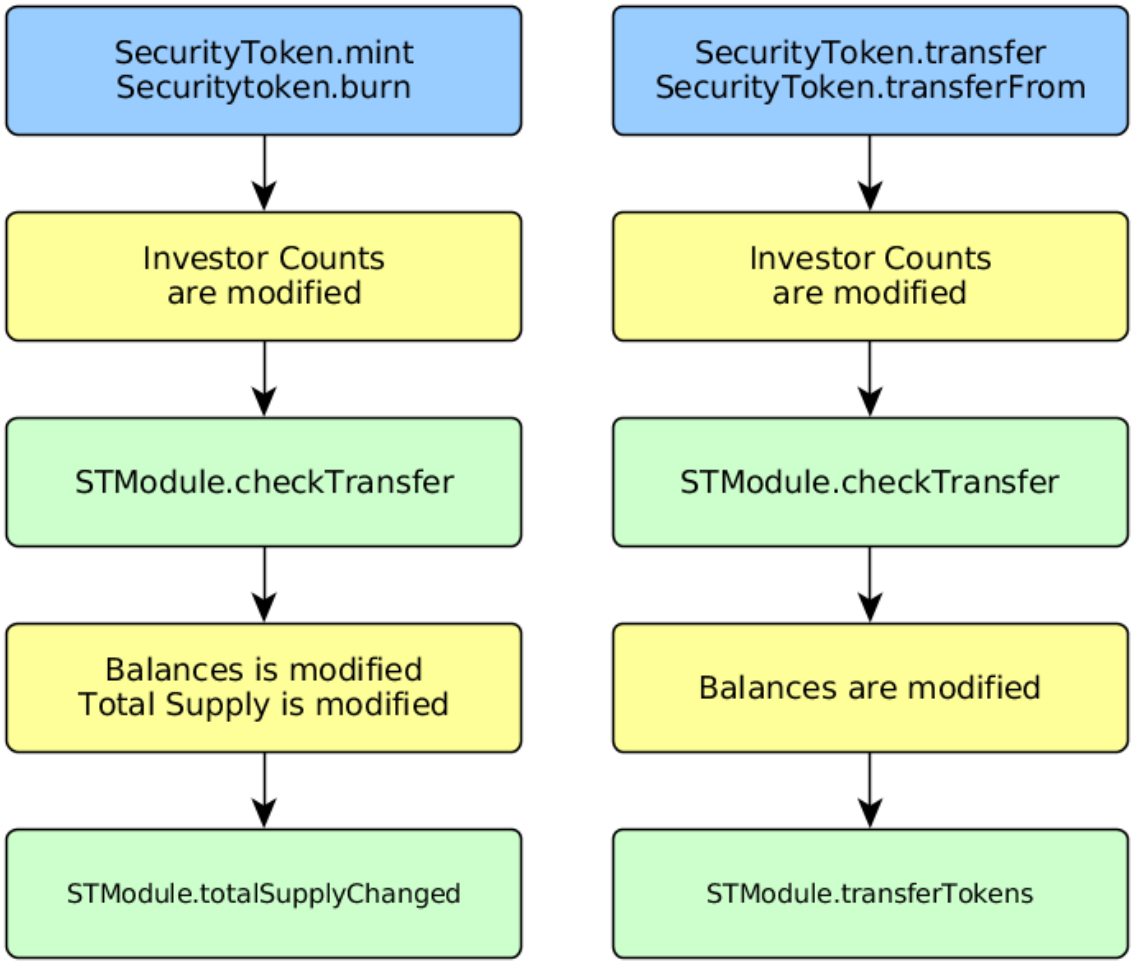

Fig. 1: SecurityToken

6.7.5 Events

Contracts that include modular functionality have the following events:

`Modular.`**`ModuleAttached`** (*address module*, *bytes4[] hooks*, *bytes4[] permissions*)
Emitted whenever a module is attached with `Modular.attachModule`.

`Modular.`**`ModuleHookSet`** (*address module*, *bytes4 hook*, *bool active*, *bool always*)
Emitted once for each hook that is set when a module is attached with `Modular.attachModule`.

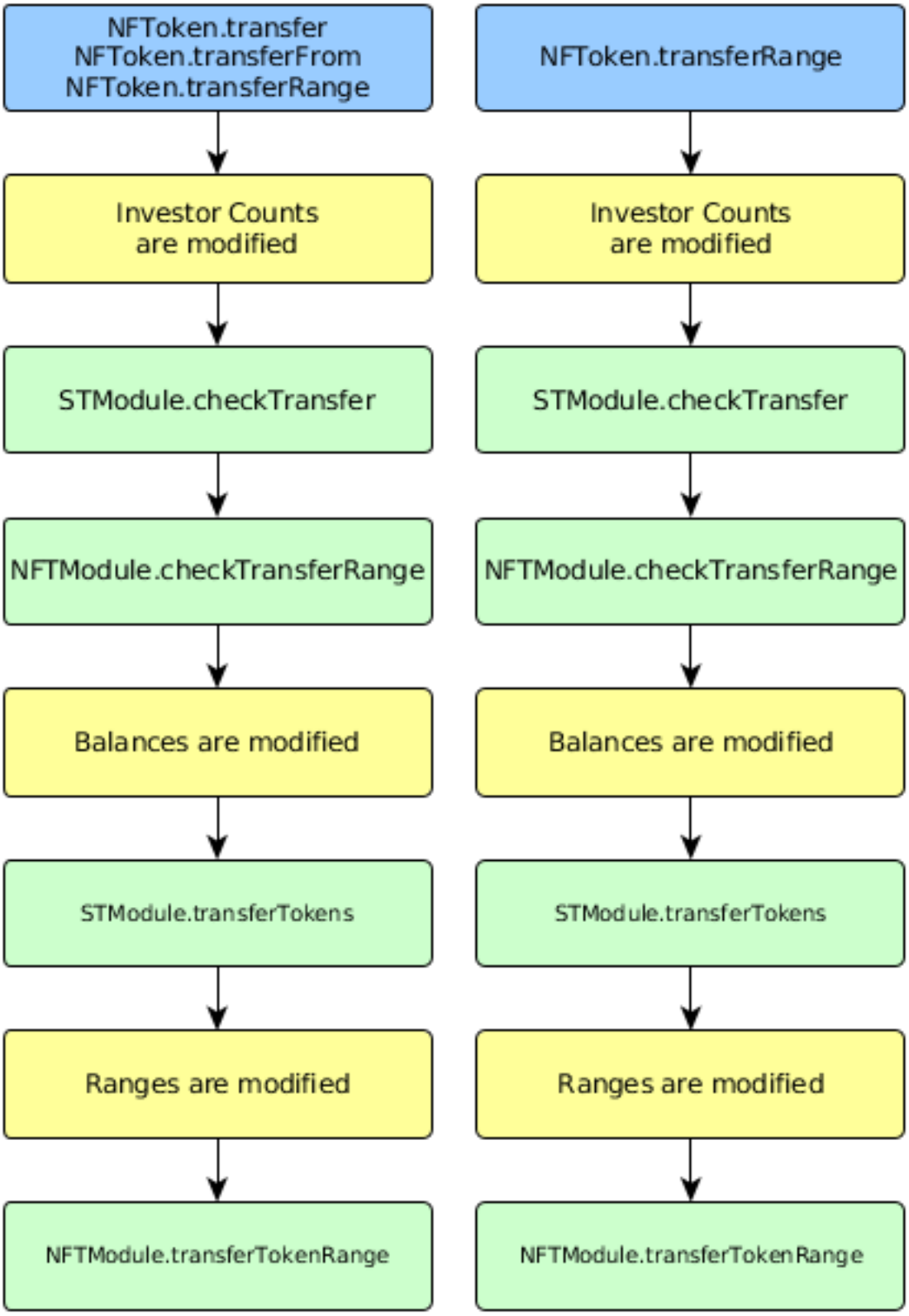

Fig. 2: NFToken

`Modular.`**`ModuleDetached`**(*address module*)
Emitted when a module is detached with `Modular.detachModule`.

6.7.6 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.

6.8 Governance

The `Goveranance` contract is a special type of module that may optionally be attached to an *IssuingEntity*. It is used to add on-chain voting functionality for token holders. When attached, it adds a permissioning check before increasing authorized token supplies or adding new tokens.

SFT includes a very minimal proof of concept as a starting point for developing a governance contract. It can be combined with a checkpoint module to build whatever specific setup is required by an issuer.

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

- Governance.sol: A minimal implementation of `Goverance`, intended for testing purposes or as a base for building a functional contract.
- IGovernance.sol: The minimum contract interface required for a governance module to interact with an `IssuingEntity` contract.

6.8.1 Public Constants

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

```
>>> governance.issuer()
0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06
```

6.8.2 Checking Permissions

The following methods must return `true` in order for the calling methods to execute.

`Governance.`**`addToken`**(*address _token*)
Called by `IssuingEntity.addToken` before associating a new token contract.

`Governance.`**`modifyAuthorizedSupply`**(*address _token*, *uint256 _value*)
Called by `TokenBase.modifyAuthorizedSupply` before modifying the authorized supply.

6.9 Investor Data Standards

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

6.9.1 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.

6.9.2 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.

6.9.3 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 bytes3: `0x414c00`

A CSV of country and region codes is available here.

6.10 Glossary

- **Authority**: A collection of one or more addresses permitted to call specific admin-level functionality in a multisig contract.

- **Custodian**: An entity that holds tokens on behalf of investors, without taking beneficial ownership. May be a natural person, legal entity, or autonomous smart contract. Examples of custodians include broker/dealers, escrow agreements and secondary markets.
- **Entity**: A participant in the SFT protocol. Entity may refer to natural persons or legal entities.
- **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 a token or custodian contract, 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

Symbols

A

B

C

D

E

G

I

M

N

O

R

S

T

U